

# Programmation, Conception et *Design Patterns* en *JavaScript* ; *Web Côté Client* avec *jQuery*

Rémy Malgouyres  
LIMOS UMR 6158, IUT, département info  
Université Clermont Auvergne  
B.P. 86  
63172 AUBIERE cedex  
<https://malgouyres.eu/>

Tous mes cours sur le *Web* sont sur le *Web* :

Cours de programmation *WEB* sur les documents hypertexte *HTML/CSS* :

<https://malgouyres.eu/programmation-html-css>

Tutoriel sur le *CMS Drupal* :

<https://malgouyres.eu/tutoriel-drupal>

Cours de programmation *WEB* côté serveur en *PHP* :

<https://malgouyres.eu/programmation-php>

Cours de programmation *WEB* côté client en *JavaScript* :

<https://malgouyres.eu/programmation-javascript>

Cours sur l'administration de serveurs (Serveurs *WEB* avec *apache*, *SSL*, *LDAP*...) :

<https://malgouyres.eu/administration-reseau>

# Table des matières

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Premiers pas en <i>JavaScript</i></b>                               | <b>6</b>  |
| 1.1      | Balise <code>&lt;script&gt;</code> et Hello world en <i>JavaScript</i> | 6         |
| 1.1.1    | <i>Hello Word</i> Avec <code>document.write</code>                     | 6         |
| 1.1.2    | <i>Hello Word</i> Avec la Méthode <code>alert</code> de <i>Popup</i>   | 6         |
| 1.1.3    | Document <i>Template</i> et Modification Dynamique de la Vue           | 7         |
| 1.2      | Types, Variables et Portée   | 8         |
| 1.3      | Fonctions  | 8         |
| 1.4      | Objets   | 10        |
| 1.4.1    | Création d'un objet au moyen d'un littéral                             | 10        |
| 1.4.2    | Méthodes   | 11        |
| 1.4.3    | Objets Imbriqués (Composites et Composés)                              | 12        |
| 1.4.4    | Constructeur d' <code>Object</code> et mot réservé <code>new</code>    | 14        |
| 1.5      | Tableaux (le type <code>Array</code> )                                 | 15        |
| 1.5.1    | Notion d' <code>Array</code> et construction                           | 15        |
| 1.5.2    | Quelques méthodes prédéfinies sur le type <code>Array</code>           | 16        |
| 1.6      | Exemple : traitement d'un formulaire avec <i>jQuery</i>                | 16        |
| 1.6.1    | Qu'est-ce que <i>jQuery</i> ?  | 16        |
| 1.6.2    | Récupérer, filtrer et afficher les données d'un formulaire             | 17        |
| <b>2</b> | <b>Programmation Fonctionnelle et Objet en <i>JavaScript</i></b>       | <b>20</b> |
| 2.1      | Passages d'Arguments, Objets <code>this</code> et <i>Pattern that</i>  | 20        |
| 2.1.1    | Passage d'arguments par objets   | 20        |
| 2.1.2    | Invocation de Méthode avec le <i>Pattern "apply"</i>                   | 21        |
| 2.1.3    | Accès au Composite à partir du Composé ( <i>pattern that</i> )         | 23        |
| 2.2      | Le <i>Pattern</i> Module   | 24        |
| 2.2.1    | Cycle de Vie d'une Variable Locale à une Fonction                      | 24        |
| 2.2.2    | Principe Général du <i>Pattern</i> Module                              | 26        |
| 2.3      | Exemple de fabrique sommaire   | 28        |
| 2.4      | Structuration d'une application  | 29        |
| 2.5      | Exemple : un module <code>metier.regexUtil</code>                      | 31        |
| 2.6      | Modélisation de Modules Métier (version 1)                             | 35        |
| 2.6.1    | Attributs et méthodes statiques (Version 1)                            | 36        |
| 2.6.2    | Exemple : Module Métier <code>adresse</code>                           | 38        |
| 2.6.3    | Fabrique Générique d'Instances Métier (Version 1)                      | 40        |
| 2.6.4    | Exemple : La Fabrique du Module <code>adresse</code>                   | 45        |
| 2.6.5    | Utilisation : Création d'un Module <code>myApp.view.adresse</code>     | 48        |
| 2.7      | Interface Générique de Objets métier                                   | 52        |

|          |  |            |
|----------|--|------------|
| 2.7.1    | Implémentation d'interfaces en <i>JavaScript</i> . . . . .                             | 52         |
| 2.7.2    | Interface des instances de modules métier . . . . .                                    | 53         |
| 2.7.3    | Exemple d'utilisation : Méthode d'affichage générique . . . . .                        | 55         |
| <b>3</b> | <b>Constructeurs, Prototype et <i>Patterns</i> Associés</b>                            | <b>59</b>  |
| 3.1      | Constructeurs . . . . .  | 59         |
| 3.2      | Prototypes . . . . .   | 60         |
| 3.2.1    | Notion de prototype . . . . .  | 60         |
| 3.2.2    | Surcharge des méthodes du prototype : l'exemple de <code>toString</code> . . . . .     | 62         |
| 3.3      | Modélisation de Modules Métier (version 2) . . . . .                                   | 63         |
| 3.3.1    | Attributs et méthodes statiques (version 2) . . . . .                                  | 63         |
| 3.3.2    | Fabrique générique d'instances métier (version 2) . . . . .                            | 65         |
| 3.3.3    | Utilisation avec l'affichage générique d'objets métier . . . . .                       | 70         |
| 3.4      | Patterns <i>pseudo-classique</i> (à éviter) . . . . .                                  | 70         |
| <b>4</b> | <b>Formulaires, Filtrage, <i>Pattern Mediator</i></b>                                  | <b>73</b>  |
| 4.1      | Filtrage Basique des Inputs d'un Formulaire . . . . .                                  | 73         |
| 4.2      | <i>Pattern Mediator</i> pour le filtrage d'attributs . . . . .                         | 75         |
| 4.3      | Exemple :<br>Génération automatique de formulaire d'adresse . . . . .                  | 78         |
| <b>5</b> | <b>Exemple d'Application avec <i>IHM</i></b>   | <b>84</b>  |
| 5.1      | Principe de l'application et analyse fonctionnelle . . . . .                           | 84         |
| 5.2      | Modèle de donnée . . . . .   | 84         |
| 5.3      | <i>Pattern Mediator</i> : centraliser les événements . . . . .                         | 87         |
| 5.4      | Événements concernant les personnes . . . . .  | 90         |
| 5.4.1    | Enregistrement des événements utilisateurs via <i>jQuery</i> . . . . .                 | 90         |
| 5.4.2    | Mise à jour du panneau des détails . . . . .   | 93         |
| 5.4.3    | Mise à jour du panneau des <i>items</i> . . . . .                                      | 94         |
| 5.4.4    | Bouton " <i>Supprimer</i> " . . . . .  | 96         |
| 5.4.5    | Bouton " <i>Modifier</i> " et affichage du formulaire . . . . .                        | 97         |
| 5.4.6    | Bouton " <i>Ajouter une personne</i> " . . . . .                                       | 98         |
| 5.4.7    | Validation du formulaire de modification . . . . .                                     | 99         |
| 5.4.8    | Validation du formulaire d'ajout d'une personne . . . . .                              | 100        |
| 5.4.9    | Code <i>HTML</i> de la vue et invocation des méthodes . . . . .                        | 101        |
| 5.5      | Événements concernant les Adresses . . . . .   | 104        |
| 5.5.1    | Enregistrement des événements utilisateurs via <i>jQuery</i> . . . . .                 | 104        |
| 5.5.2    | Boutons d'ajout, de suppression, et de modification . . . . .                          | 106        |
| 5.5.3    | Création d'une nouvelle adresse . . . . .  | 109        |
| 5.5.4    | Modification d'une adresse . . . . .   | 110        |
| <b>6</b> | <b>Requêtes Asynchrones et <i>API Restful</i></b>                                      | <b>113</b> |
| 6.1      | Qu'est-ce qu'une requête asynchrone ? . . . . .  | 113        |
| 6.2      | Requêtes <i>Ajax</i> . . . . .   | 114        |
| 6.3      | Qu'est-ce qu'une <i>API REST</i> (ou systèmes <i>Restful</i> ) ? . . . . .             | 117        |
| 6.4      | Persistance par Requêtes sur une <i>API Restful</i> . . . . .                          | 118        |
| 6.4.1    | Création du Module <code>persistance</code> et Objet <code>statusCode</code> . . . . . | 118        |

|          |   |            |
|----------|---|------------|
| 6.4.2    | Construction du modèle à partir de la base de données . . . . . | 119        |
| 6.4.3    | Création, Mise à jour, et suppression des personnes . . . . .   | 123        |
| 6.4.4    | Création, Mise à jour, et suppression des adresses . . . . .    | 126        |
| <b>A</b> | <b>Graphisme avec les Canvas <i>HTML5</i></b>                   | <b>130</b> |
| A.1      | Notion de <i>canvas</i> . . . . .                               | 130        |
| A.2      | Exemple d'animation dans un <i>canvas</i> . . . . .             | 131        |
| <b>B</b> | <b>Programmation Événementielle en <i>JavaScript</i></b>        | <b>133</b> |
| B.1      | Rappel sur la Gestion d'Événements en <i>CSS</i> . . . . .      | 133        |
| B.2      | Événements en <i>Javascript</i> . . . . .                       | 134        |
| B.2.1    | Le principe des événements en <i>Javascript</i> . . . . .       | 134        |
| B.2.2    | Exemple de mise à jour d'un élément . . . . .                   | 135        |
| B.2.3    | Formulaires Dynamiques an <i>Javascript</i> . . . . .           | 136        |
| <b>C</b> | <b>Gestion des fenêtres</b>                                     | <b>139</b> |
| C.1      | Charger un nouveau document . . . . .                           | 139        |
| C.2      | Naviguer dans l'historique . . . . .                            | 140        |
| C.3      | Ouvrir une nouvelle fenêtre (popup) . . . . .                   | 141        |
| <b>D</b> | <b><i>Document Object Model (DOM)</i></b>                       | <b>142</b> |
| D.1      | Qu'est-ce que le <i>DOM</i> ? . . . . .                         | 142        |
| D.2      | Sélection et Manipulation de Base sur le <i>DOM</i> . . . . .   | 143        |
| D.2.1    | Sélection de tout ou partie des éléments . . . . .              | 143        |
| D.2.2    | Filtrage par le texte . . . . .                                 | 144        |
| D.2.3    | Application de Méthode aux éléments . . . . .                   | 145        |
| D.2.4    | Événements et <i>Callbacks</i> . . . . .                        | 146        |
| D.2.5    | Fitrage d'un Tableau . . . . .                                  | 148        |

# Architectures client/serveur et *API*

## Architecture d'une application multi plate-formes

Une application multi plate-formes contemporaine cherchera à se structurer suivant (voir figure 1) :

1. Une application sur un serveur (*API*) qui traitera les données et assurera la persistance ;
2. Une application sur chaque type de client, qui utilise ce serveur via des requêtes, et gère l'interface (par exemple une *Interface Homme Machine (IHM)*).

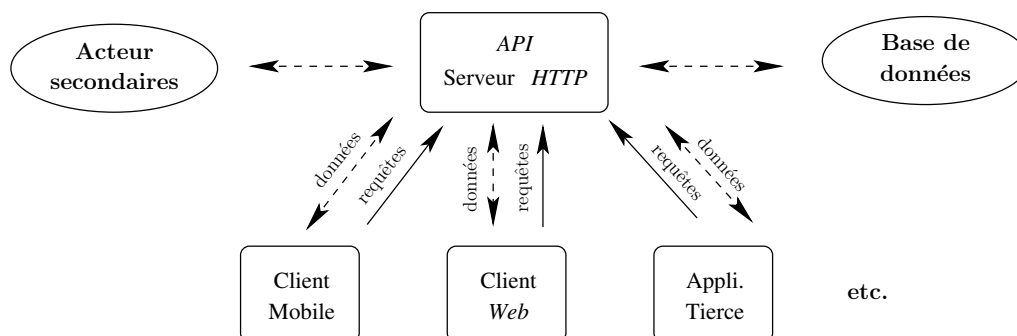


FIGURE 1 : La structure typique d'une application multi plate-formes

On aura dans la mesure du possible intérêt à limiter le plus possible le travail côté client pour les raisons suivantes :

1. L'implémentation côté client dépend de la plate-forme et l'implémentation sur le serveur constitue une forme de *factorisation* du code.
2. Sur certaines plate-formes, comme dans le cas des applications web en *JavaScript*, la sécurité et la confidentialité côté client sont très mauvaises, alors que nous pouvons implémenter la sécurité côté serveur.

Cependant, dans la mesure du possible, les opérations peu sensibles, par exemple concernant l'ergonomie, se feront côté client pour limiter les coûts d'infrastructure (nombre de serveurs...) et améliorer la réactivité de l'application.

## Le cas de l'application *Web*

Dans ce cours, nous étudions le développement d'applications *Web* (auxquelles on accède via un navigateur internet), avec une architecture client/serveur dans laquelle (voir la figure 2) :

- Notre *API* est un serveur *HTTP* implémenté en *PHP* avec une architecture *MVC* et *DAL*;
- Notre application côté client est en *JavaScript* (qui s'est imposé comme un langage standard côté client), et utilise la librairie *jQuery* pour la gestion des événements, des vues, et des interactions (requêtes et échange de données au format *JSON*) avec le serveur.

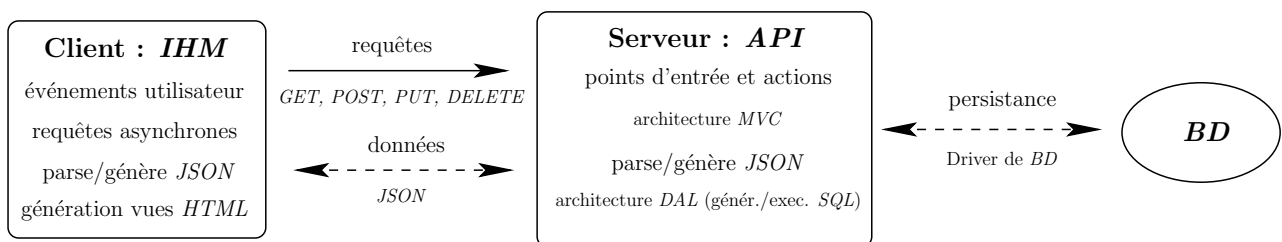


FIGURE 2 : L'architecture *client/serveur* de notre application *Web*

# Chapitre 1

## Premiers pas en *JavaScript*

### 1.1 Balise `<script>` et Hello world en *JavaScript*

#### 1.1.1 *Hello Word Avec document.write*

Une première manière d'insérer un script *JavaScript* dans un fichier *HTML* est de mettre le code *JavaScript* dans une balise `<script></script>`. On a alors accès au document dans le code *JavaScript* et on peut sortir du code *HTML* :

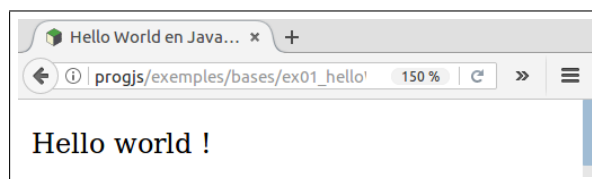


FIGURE 1.1 : Illustration du code source 1.1

Code Source 1.1 : `/bases-js/ex01-helloWorldTest.html` (cf. Fig 1.1)

```
1 <!doctype HTML>
2 <html lang="fr">
3 <head>
4 <meta charset="UTF-8" />
5 <title>Hello World en Javascript</title>
6 </head>
7 <body>
8   <p>
9     <script>
10       document.write("Hello world !");
11     </script>
12   </p>
13 </body>
14 </html>
```

#### 1.1.2 *Hello Word Avec la Méthode alert de Popup*

Une autre manière d'insérer un script *JavaScript* dans un fichier *HTML* est de mettre le code *JavaScript* dans un fichier `.js` séparé, qui est inclus dans le *HTML* au niveau du header par une balise `<script src='...'></script>`.

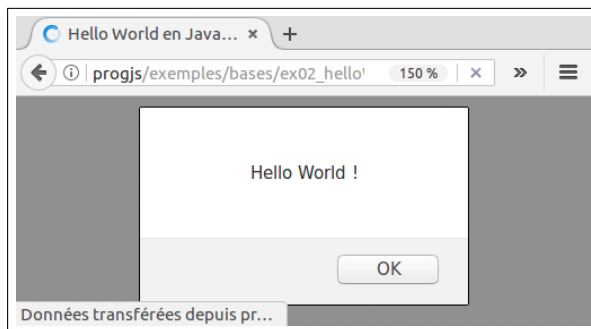


FIGURE 1.2 : Illustration du code source 1.2

Code Source 1.2 : /bases-js/ex02-helloWorldTest.html (cf. Fig 1.2)

```

1 <!doctype HTML>
2 <html lang="fr">
3 <head>
4   <meta charset="UTF-8" />
5   <title>Hello World en Javascript</title>
6   <p>
7     <script src="./ex02-helloWorld.js"></script>
8   </p>
9 </head>
10 <body>
11 </body>
12 </html>

```

Code Source 1.3 : /bases-js/ex02-helloWorld.js

```

1 alert("Hello World !");

```

Dans ce dernier cas, on ne peut pas générer de code directement dans le document *HTML* (avec `document.write`) dans le fichier *JavaScript*, mais il y a d'autres avantages (factorisation et mise en cache du code *JavaScript* partagé entre plusieurs pages *HTML* par exemple).

La fonction `alert` peut par exemple utilisé pour du débogage si l'on ne dispose pas d'un outil de *debug* intégré dans notre *IDE*.

### 1.1.3 Document *Template* et Modification Dynamique de la Vue

Dans une application *HTML/JavaScript* côté client, on organise souvent le code avec, d'une part, un squelette de document *HTML*, appelé *template*, et d'autre part des fichiers *JavaScript*, inclus le plus tard possible dans la page *HTML*, qui modifie *a posteriori* le document pour générer certaines parties dynamiquement.

Code Source 1.4 : /bases-js/ex02bis-helloWorldTemplateTest.html

```

1 <!doctype HTML>
2 <html lang="fr">
3 <head>
4   <meta charset="UTF-8" />
5   <title>Hello World en Javascript</title>
6 </head>
7 <body>

```



```

8  <!-- Création d'une page "Template", squelette de notre page -->
9  <!-- Paragraphe qui va contenir le code -->
10 <p id="paragrapheResultat"></p>
11 <!-- Inclusion du fichier JavaScript pour le code dynamique -->
12 <script src="/ex02bis-helloWorldTemplate.js"></script>
13 </body>
14 </html>

```

Code Source 1.5 : /bases-js/ex02bis-helloWorldTemplate.js

```

1 // Génération dynamique de code HTML :
2 var codeHTML = "Hello world !";
3 // Mise à jour de la vue
4 document.getElementById("paragrapheResultat").innerHTML = codeHTML;

```

## 1.2 Types, Variables et Portée

Le *JavaScript* est un langage faiblement typé, car on n'indique pas le type des variables lors de la déclaration. Lors de la déclaration des variables, le type est fixé implicitement par le type de la donnée affectée à la variable.

La déclaration de la variable peut contenir ou non le mot clef **var**. Une variable déclarée avec le mot clef **var** (on parle de déclaration *explicite*) est locale à la fonction où la variable est déclarée. Une variable déclarée sans le mot clef **var** (on parle de déclaration *implicite*) est globale.

Il n'y a pas, contrairement au *C++* ou *Java*, de visibilité locale à un bloc. Une variable déclarée n'importe où dans une fonction est visible dans toute la fonction au moins. Pour cette raison, on déclarera systématiquement les variables locales à la fonction au début du corps de la fonction, contrairement aux bonnes pratiques dans d'autres langages où on déclare la variable au plus près de son point de première utilisation.

Dans les programmes assez gros structurés en modules ou *packages*, on peut créer en *JavaScript* l'équivalent d'un *namespace* par un patron de conception consistant à mettre le code de l'ensemble d'un module dans le corps de définition d'une fonction ou dans un littéral définissant un objet (voir plus loin pour la notion d'objet).

## 1.3 Fonctions

Les fonctions en *JavaScript* sont déclarées par le mot clef **function**. c'est un type de données comme un autre, et une fonction peut ainsi être affectée à une variable. Voici un exemple de fonction qui calcule le prix *TTC* d'un produit à partir de son prix hors taxes. Comme les paramètres des fonctions ne sont pas typés, on peut vérifier le type des paramètres dans la fonction et éventuellement renvoyer une exception si le type du paramètre effectif n'est pas le bon.

Code Source 1.6 : /bases-js/ex03-functionTest.html

```

1 <!doctype HTML>
2 <html lang="fr">
3 <head>
4 <meta charset="UTF-8" />

```

```

5 <title>Fonctions</title>
6 </head>
7 <body>
8   <!-- Création d'une page "Template", squelette de notre page -->
9   <!-- Paragraphe qui va contenir le code -->
10  <p id="paragrapheResultat"></p>
11  <!-- Inclusion du fichier JavaScript pour le code dynamique -->
12  <script src="/ex03-function.js"></script>
13 </body>
14 </html>

```

Code Source 1.7 : /bases-js/ex03-function.js

```

1  /** @description Calcule le prix TTC d'un produit
2   * @function calculPrixTTC
3   * @param {number} prixHT - Le prix hors taxe du produit
4   * @param {number} tauxTVA - Le taux de TVA à appliquer
5   * @throws Si les paramètres prixHT et tauxTVA ne sont pas des nombres
6   */
7  var calculPrixTTC = function (prixHT, tauxTVA){
8    if (!(typeof prixHT === "number") || !(typeof tauxTVA === "number")){
9      throw new Error("Function calculPrixTTC appelée avec paramètre incorrect.")
10    }
11    return prixHT*(1.0+tauxTVA/100.0);
12  };
13
14
15  // Génération dynamique de code HTML :
16  var codeHTML = "";
17  // Appel correct de la fonction :
18  try{
19    codeHTML += "Prix TTC : " + calculPrixTTC(180.0, 19.6);
20  }catch (err){
21    alert(err);
22  }
23  // Appel incorrect de la fonction déclenchant une exception :
24  try{
25    codeHTML += "Prix TTC : " + calculPrixTTC("coucou", 19.6);
26  }catch (err){
27    alert(err);
28  }
29  // Mise à jour de la vue
30  document.getElementById("paragrapheResultat").innerHTML = codeHTML;

```

Notons que l'on peut aussi déclarer une fonction un peu comme en *PHP* de la manière suivante :

Code Source 1.8 : Ne jamais faire ça !

```

1  function myFunction(myParam){
2    return (myParam < 0);
3  }

```

mais la fonction est alors globale (son nom existe dans tout le programme).



La **bonne pratique** consiste à déclarer les éléments d'un programme de sorte qu'ils aient la portée la plus locale possible, donc à déclarer la fonction avec le mot clé `var` comme dans le premier exemple de fonction ci-dessus.

## 1.4 Objets

Un *objet JavaScript* rassemble plusieurs propriétés, qui peuvent être des données, d'autres objets, ou encore des fonctions, alors appelées *méthodes*. Un objet n'est ni tout à fait une structure comme en *C*, ni tout à fait une classe comme dans un *langage objet classique*. Par exemple, un objet *JavaScript* n'a pas de visibilité (privée, public) pour ses propriétés. Par ailleurs, le principal mécanisme d'héritage en *JavaScript* se fait par la notion de *prototype* et est très différent de l'héritage dans les langages objet classiques. Là encore, on peut mimer une notion de visibilité via des patrons de conception.

Les noms de propriétés peuvent être

- Soit une chaîne de caractère (comme "nom de propriété!") quelconque (respecter les doubles *quotes* dans un tel cas).
- Soit des noms légaux (commençant par une lettre suivi par une suite de lettres, chiffres, et *underscores* (caractère `_`) auquel cas les doubles *quotes* sont optionnelles pour désigner le nom.

### 1.4.1 Création d'un objet au moyen d'un littéral

On peut créer un nouvel objet par un littéral, en définissant ses propriétés des accolades `{}`. On met alors chaque nom de propriété suivi d'un `:` suivi de la valeur de la propriété. Les propriétés ainsi construites sont séparées par des virgules.

Code Source 1.9 : `/bases-js/ex04-objectLiteral.js`

```

1  /** @description Littéral définissant un objet appelé "produit" */
2  var produit = {
3      "denomination" : "Notebook sous Ubuntu 4 cores 2.0GB",
4      "prixHT" : 180.0,
5      "tauxTVA" : 19.6
6  };
7  /**
8   * @description Calcule le prix TTC d'un produit
9   * @function calculPrixTTC
10  * @param {Object} prod - Le produit à traiter
11  * @param {number} prod.prixHT - Le prix hors taxe du produit
12  * @param {number} prod.tauxTVA - Le taux de TVA à appliquer
13  * @throws Si les propriétés du paramètre prod ne sont pas des nombres
14  */
15  var calculPrixTTC = function(prod){
16      // Test d'existence des propriétés de l'objet :
17      if ("prixHT" in prod && "tauxTVA" in prod){
18          return prod.prixHT*(1.0+prod.tauxTVA/100.0);
19      }else{
20          // Rejet d'une exception personnalisée :
21          // On rejette un objet avec une prop. "name" et une prop. "message".
22          throw {

```

```

23     name : "Bad Parameter",
24     message : "Mauvais type de paramètre pour la fonction calculPrixTTC"
25   };
26 }
27 };
28
29 // Essai d'appel de la fonction
30 try{
31   var codeHTML = "Prix TTC du produit \""+produit.denomination
32                 + "\" : "+calculPrixTTC(produit);
33 } catch (e) { // affichage de l'exception personnalisée.
34   alert("Une erreur \""+ e.name + "\" s'est produite :\n" + e.message);
35 }
36 // Mise à jour de la vue
37 document.getElementById("paragrapheResultat").innerHTML = codeHTML;

```

### 1.4.2 Méthodes

Un objet peut contenir des propriétés qui sont de type **function**. On parle alors de *méthode* de l'objet. Dans une méthode, on accède aux propriétés de l'objet grâce à l'identificateur **this**, désignant l'objet auquel appartient la méthode.

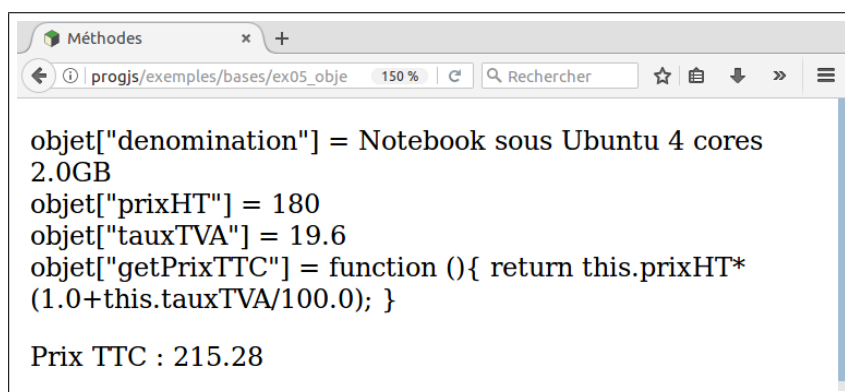


FIGURE 1.3 : Illustration du code source 1.10

Code Source 1.10 : /bases-js/ex05-objectMethod.js (cf. Fig 1.3)

```

1  /** @description Littéral définissant un objet appelé "produit" */
2  var produit = {
3    "denomination" : "Notebook sous Ubuntu 4 cores 2.0GB",
4    "prixHT" : 180.0,
5    "tauxTVA" : 19.6,
6    /**
7     * @description Calcule le prix TTC d'un produit
8     * @method calculPrixTTC
9     * @return Le prix TTC du produit
10    */
11    getPrixTTC : function() {
12      return this.prixHT*(1.0+this.tauxTVA/100.0);
13    }
14  };
15

```

```

16 // Fonction dans le contexte global :
17 /** @description Génère le code HTML pour afficher un objet quelconque
18  * @function getHtmlObjet
19  * @param {Object} objet - L'objet à traiter
20  * @return Le code HTML pour afficher l'objet
21  */
22 var getHtmlObjet = function(objet){
23     var chaine = "";
24     // Parcours de toutes les propriétés de l'objet (style "foreach") :
25     for (var nom in objet){
26         chaine += "objet[\""+nom+"\"] = "
27                 + objet[nom] // Appel de la méthode toString par défaut
28                 + "<br/>";
29     }
30     return chaine;
31 };
32
33 // appel d'une fonction définie dans le contexte global :
34 var codeHTML = "<p>" + getHtmlObjet(produit) + "</p>";
35 // appel d'une méthode :
36 codeHTML += "<p>Prix TTC : " + produit.getPrixTTC() + "</p>";
37 // Mise à jour de la vue
38 document.getElementById("paragrapheResultat").innerHTML = codeHTML;

```

Une méthode d'objet *JavaScript* n'est pas tout à fait comme une méthode d'un langage à objet classique, car la méthode *JavaScript* existe en autant d'exemplaires qu'il y a d'instance des objets. Nous verrons plus loin la notion de *prototype*, qui permet de créer des méthodes qui existent en un seul exemplaire pour toute une classe d'objets ayant les mêmes propriétés.

### 1.4.3 Objets Imbriqués (Composites et Composés)

Il est possible, dans un littéral d'objet, de créer une propriété qui est elle-même un objet. On peut parler de *composite* pour l'objet qui contient un autre objet, qui est alors un *composé*.

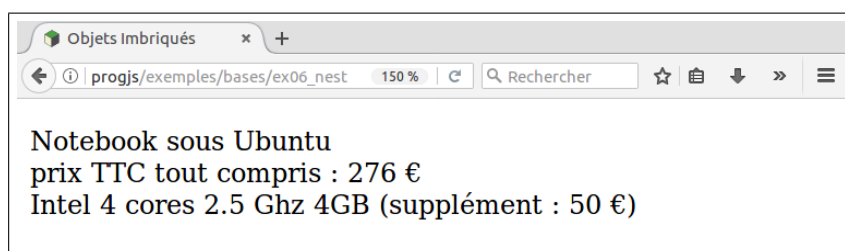


FIGURE 1.4 : Illustration du code source 1.11

Code Source 1.11 : `/bases-js/ex06-nestedObjects.js` (cf. Fig 1.4)

```

1 /** @description Littéral définissant un objet appelé "produit" */
2 var produit = {
3     denomination : "Notebook sous Ubuntu",
4     prixHT_base : 180.0,
5     tauxTVA : 20.0,
6     /** @description Objet "niché" dans un sur-objet (Options du produit) */
7     options : {
8         processor : "Intel 4 cores 2.5 Ghz",

```

```

9      memory : "4GB",
10     "prix supplémentaire HT" : 50.0,
11     /** @description Génère le code HTML des options
12      * @method getHTML
13      */
14     getHtml : function(){
15         return this.processor + " " + this.memory +
16             " (supplément : " + this["prix supplémentaire HT"] + " €euro;)";
17     },
18 },
19 /** @description Génère le code HTML complet du produit
20  * @method getHTML
21  */
22 getHtml : function(){
23     return this.denomination +
24         "<br/>prix TTC tout compris : "
25         + (this.prixHT_base + (this.options["prix supplémentaire HT"] || 0.0)
26           )
27         * (1.0 + this.tauxTVA / 100.0)
28         + " €euro ;<br/>" + this.options.getHtml() + "<br/>";
29 }
30 };
31 // appel d'une méthode :
32 var codeHTML = "<p>" + produit.getHtml() + "</p>";
33 // Mise à jour de la vue
34 document.getElementById("paragrapheResultat").innerHTML = codeHTML;

```

On peut aussi choisir de définir l'objet contenant les options dans une méthode `getOptions` du composite, qui retourne un objet. La méthode `getOptions` joue alors un rôle de “fabrique”.

Code Source 1.12 : `/bases-js/ex06-objectReturnedInMethod.js`

```

1  /** @description Littéral définissant un objet appelé "produit" */
2  var produit = {
3      denomination : "Notebook sous Ubuntu",
4      prixHT_base : 180.0,
5      tauxTVA : 20.0,
6      /** @description Construction d'un objet dans une méthode
7       *
8       */
9      getOptions : function(){
10         // Création de l'objet option dans la fonction ("fabrique")
11         return {
12             processor : "Intel 4 cores 2.5 Ghz",
13             memory : "4GB",
14             "prix supplémentaire HT" : 50.0,
15             /** @description Génère le code HTML des options
16              * @method getHTML
17              */
18             getHtml : function(){
19                 return this.processor + " " + this.memory +
20                     " (supplément : " + this["prix supplémentaire HT"] + " €euro;)";
21             }
22         };
23     },
24     /** @description Génère le code HTML complet du produit
25     * @method getHTML

```

```

25  */
26  getHtml : function() {
27      return this.denomination +
28          "<br/>prix TTC tout compris : "
29          + (this.prixHT_base
30            + (this.getOptions()[ "prix supplémentaire HT" ] || 0.0))
31            *(1.0+this.tauxTVA/100.0)
32            + " €euro ;<br/>" + this.getOptions().getHtml() + "<br/>";
33  }
34  };
35
36  // appel d'une méthode :
37  var codeHTML = "<p>" + produit.getHtml() + "</p>";
38  // Mise à jour de la vue
39  document.getElementById( "paragrapheResultat" ).innerHTML = codeHTML ;

```

#### 1.4.4 Constructeur d'Object et mot réservé new

On peut créer un objet via le constructeur `Object`. Voici un exemple où l'on crée un objet qui représente un produit. On crée ensuite une fonction qui calcule le prix *TTC* de ce produit après avoir testé l'existence d'attributs.

Code Source 1.13 : /bases-js/ex07-objectNew.js

```

1  /** @description Instantiation d'un Objet "produit" */
2  var produit = new Object() ;
3  // Ajout dynamique de propriétés
4  produit.denomination = "Notebook sous Ubuntu 4 cores 2.0GB";
5  produit.prixHT = 180.0 ;
6  produit.tauxTVA = 20.0 ;
7
8  /**
9   * @description Calcule le prix TTC d'un produit
10  * @function calculPrixTTC
11  * @param {Object} prod - Le produit à traiter
12  * @param {number} prod.prixHT - Le prix hors taxe du produit
13  * @param {number} prod.tauxTVA - Le taux de TVA à appliquer
14  * @throws Si les propriétés du paramètre prod ne sont pas des nombres
15  */
16  var calculPrixTTC = function(prod){
17      if ( "prixHT" in prod && "tauxTVA" in prod ){
18          return prod.prixHT*(1.0+prod.tauxTVA/100.0) ;
19      } else {
20          throw new Error( "Mauvais type de paramètre pour la fonction calculPrixTTC" );
21      }
22  }
23  // Génération de code HTML
24  var codeHTML = "Prix TTC du produit \""+produit.denomination+"\" : "+
25      calculPrixTTC(produit) ;
26  // Mise à jour de la vue
27  document.getElementById( "paragrapheResultat" ).innerHTML = codeHTML ;

```

Dans la mesure du possible, il est préférable de définir les objets *JavaScript* par des littéraux car ça peut être plus efficace que la construction dynamique avec le constructeur `Object`.

## 1.5 Tableaux (le type Array)

### 1.5.1 Notion d'Array et construction

Dans les langages classique, un tableau est une séquence d'éléments, contigus en mémoire, avec un accès aux éléments par un indice entier. En *JavaScript*, les tableaux sont des objets dont les propriétés sont automatiquement nommées avec les chaînes '0', '1', '2'. Ces tableaux possèdent certains avantages des objets, comme par exemple la possibilité d'avoir des éléments de types différents, mais sont significativement plus lents que les tableaux classiques.

Un tableau peut être créé par un littéral, entre crochets [ ].

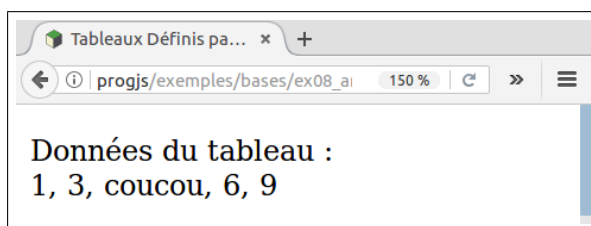


FIGURE 1.5 : Illustration du code source 1.14

Code Source 1.14 : /bases-js/ex08-arrayLitterals.js (cf. Fig 1.5)

```

1  /** @description Déclaration d'un Array sous forme de littéral */
2  var tab = [1, 3, "coucou", 6];
3  tab[4]=9; // Ajout d'un élément
4  // Génération dynamique de code HTML :
5  var codeHTML = "Données du tableau <br/>";
6  // Parcours du tableau avec un indice numérique
7  for (var i=0 ; i<tab.length ; i++){
8      if (i > 0){
9          codeHTML += ", "
10     }
11     codeHTML += tab[i];
12 }
13 // Mise à jour de la vue
14 document.getElementById("paragrapheResultat").innerHTML = codeHTML;

```

Un tableau peut aussi être créé par le constructeur d'*Array*. Celui-ci peut prendre en argument soit le nombre de cases du tableau à allouer, soit les éléments du tableau initialisés lors de la création du tableau. On peut toujours ajouter des éléments au tableau par une simple affectation de ces éléments et la mémoire évolue automatiquement.

Code Source 1.15 : /bases-js/ex09-arraysNew.js

```

1  /** @description Déclaration d'un Array sous forme de littéral */
2  var tab = new Array(1, 3, "coucou", 6);
3  tab[4]=9; // Ajout d'un élément
4  // Génération dynamique de code HTML :
5  var codeHTML = "Données du tableau <br/>";
6  // Parcours du tableau avec un indice numérique
7  for (var i=0 ; i<tab.length ; i++){
8      if (i > 0){
9          codeHTML += ", "
10     }

```



```
11 codeHTML += tab[i];  
12 }  
13 // Mise   jour de la vue  
14 document.getElementById("paragrapheResultat").innerHTML = codeHTML;
```

De m me que pour les objets, il est pr f rable de d finir les tableaux *JavaScript* par des litt raux car  a peut  tre plus efficace que la construction dynamique avec le constructeur `Array`.

### 1.5.2 Quelques m thodes pr d finies sur le type `Array`

Consid rons un tableau `array` obtenu par invocation du constructeur d'`Array` :

Code Source 1.16 :

```
1 var array = new Array(...);
```

On peut manipuler le tableau avec des m thodes de collection de conception classique :

- Suppression du dernier  l ment (l' l ment supprim  est retourn  par la m thode) :  
`function array.pop();`
- Suppression du premier  l ment (l' l ment supprim  est retourn  par la m thode) :  
`function array.shift();`
- Suppression d'une partie des  l ments :  
`function array.splice(firstElementKey, numberOfElementsToRemove);`
- Ajout d'un ou plusieurs  l ment(s)   la fin :  
`function array.push(element1, element2...);`
- Tri d'un tableau : `function array.sort(compareFuntion);`  
o  `compareFuntion` est une fonction permettant de comparer deux  l ments du tableau qui a pour prototype : `function compareFuntion(a, b);`  
et renvoie 0 si les  l ments sont  gaux, un nombre n gatif si `a` est strictement inf rieur   `b`, et un nombre positif si `a` est strictement sup rieur   `b`.  
Pour les cha nes de caract re, on peut utiliser la m thode `string.localCompare(that)` (similaire   `strcmp`).

## 1.6 Exemple : traitement d'un formulaire avec *jQuery*

### 1.6.1 Qu'est-ce que *jQuery* ?

La librairie *jQuery*, dont on peut obtenir le code et la documentation sur [api.jquery.com](http://api.jquery.com), permet de simplifier la gestion de diff rents aspects d'une application c t  client en *JavaScript* :

- Gestion des  v nements utilisateur ;
- R cup ration des valeurs saisies dans un formulaire ;
- Manipulation du document via le *DOM* ;

- Requêtes asynchrones (transfert de données entre serveur et client en dehors du chargement initial de la page) ;
- Codage des données pour la transfert (par exemple *JSON*).

Pour utiliser *jQuery*, il suffit d'insérer son code dans un script, via une balise (remplacer *x.xx.x* par le numéro de version utilisé sur [jquery.com](http://jquery.com)) :

```
<script src="https://code.jquery.com/jquery-x.xx.x.js"></script>
```

Pour travailler *offline*, on peut utiliser *jQuery* en local après téléchargement dans le répertoire courant :

```
<script src="./jquery-x.xx.x.js"></script>
```

Les méthodes de *jQuery* peuvent être appelées (avec des argument *args*) par l'abréviation *\$(args)*.

### 1.6.2 Récupérer, filtrer et afficher les données d'un formulaire

Le script suivant récupère les données d'un formulaire, les filtre par expressions régulières, et les affiche en modifiant le *DOM*.

Plus précisément, le script réalise les opération suivantes :

- Déclaration d'un gestionnaire (fonction *afficheDonneesForm*) de l'événement *submit* du formulaire ayant *formStudentData* pour *ID* ;
- Dans cette fonction *afficheDonneesForm*,
  - Récupération des valeurs saisies dans les éléments ayant *nom* et *annee* pour *ID*, qui sont respectivement un *input* et un *select*.
  - Test sur la forme (expression régulière, champs obligatoire,...) sur les valeurs des champs *nom* et *année* du formulaire (à l'aide d'un littéral de type expression régulière entre slashes */.../*).
  - Ajout dans le *<span>* ayant *spanResultat* pour *ID* du code *HTML* du résultat de la saisie (affichage du nom et de l'année, ou le cas échéant un message d'erreur).
  - Empêcher le comportement par défaut de la soumission du formulaire (appel du script *action* côté serveur lors du *click* sur l'*input* de type *submit*).

Voici le fichier *HTML* :

Code Source 1.17 : */bases-js/ex10-jQueryFormTest.html* (cf. Fig 1.6)

```
1 <!doctype HTML> <!-- Template de Document HTML -->
2 <html lang="fr">
3 <head>
4 <meta charset="UTF-8" />
5 <title>Formulaires avec jQuery</title>
6 </head>
7 <body>
8   <form id="formStudentData">
9     <p>
```

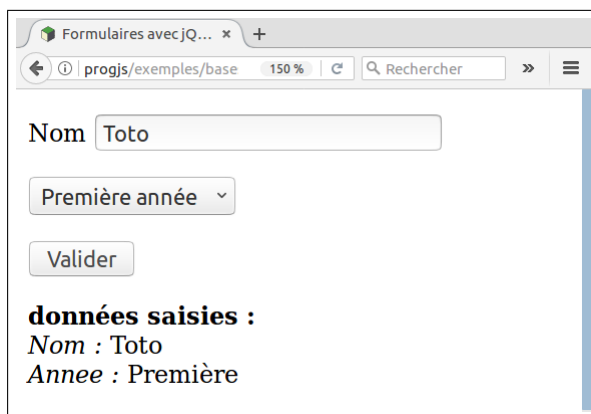


FIGURE 1.6 : Illustration du code source 1.17

```

10     <label for="nom" >Nom</label>
11     <input name="nom" id="nom"/>
12 </p>
13 <p>
14     <select name="annee" id="annee">
15         <option value="choisissez" selected disabled>-- choisissez --</option>
16
17         <option value="Premi re">Premi re ann e</option>
18         <option value="Deuxi me">Deuxi me ann e</option>
19     </select>
20 </p>
21 <p>
22     <input type="submit" value="Valider"/>
23 </p>
24 </form>
25
26 <p>
27     <strong>donn es saisies    </strong><br/>
28     <!-- Les r sultats de la saisie vont s'afficher dans ce span -->
29     <span id="spanResultat"></span>
30 </p>
31 <script src="jquery.js"></script>
32 <script src="ex10-jQueryForm.js"></script>
33 </body>
34 </html>

```

Il est recommand  de mettre, dans la mesure du possible, le script   la fin du document, car cela limite le co t et le d lai des chargements et *parsing* de la librairie *jQuery* lors d'un premier chargement (ou rafra chissement) de la page. Voici le fichier *JavaScript* :

Code Source 1.18 : /bases-js/ex10-jQueryForm.js

```

1  /** @description Callback : R cup re et affiche les inputs de formulaire.
2   *      L'impl mentation utilise jQuery.
3   * @function afficheDonneesForm
4   * @param {jQueryEvent} event - l' v nement (de type submit) g r  par ce Handler
5   */
6  var afficheDonneesForm = function(event){
7      // r cup ration (via jQuery) de la valeur de l'input d'ID "nom"
8      var nom = $("#nom").val();

```

```

9 // récupération (via jQuery) de la valeur du select d'ID "annee"
10 var annee = $("#annee").val();
11
12 // test de champs obligatoires et d'expressions régulières sur le nom
13 if ((annee !== "Première" && annee !== "Deuxième") ||
14     !/^[a-zA-ZÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖ×ØÙÚÛÜÝÞßàáâãäåæ,èéêëìíîïðñòóôõö÷øùúûÄäÜüÝýÿ
15         \s"'-]{1,}$ /
16     .test(nom)) {
17     $("#spanResultat").html("Problème : forme d'un champs incorrect.");
18 } else {
19     $("#spanResultat").html("<em>Nom : </em>" + nom + "<br />" +
20         "<em>Annee : </em>" + annee );
21 }
22 // Éviter d'appeler l'"action" par défaut (requête sur un script PHP, etc.)
23 // du formulaire lors du click sur le bouton submit
24 event.preventDefault();
25
26 // Gestion de l'événement submit du formulaire.
27 // On définit afficheDonneesForm comme gestionnaire (handler)
28 // de l'événement.
29 $("#formStudentData").on("submit", afficheDonneesForm);

```

## Chapitre 2

# Programmation Fonctionnelle et Objet en *JavaScript*

On distingue en *JavaScript* deux catégories de *patterns* (et éventuellement des *patterns* hybrides) :

- Les *patterns* dits *fonctionnels* s'appuient sur les aspects de *JavaScript* en tant que *langage fonctionnel*. Autrement dit, ces *patterns* exploitent les propriétés des fonctions *JavaScript* en tant que données, ainsi que la portée des variables dans ces fonctions.
- Les *patterns* dits *prototypaux* s'appuient sur les aspects de *JavaScript* en tant que *langage prototypal*. Ceci est lié à une propriété que possèdent tous les objets *JavaScript*, appelée le **prototype**. Le prototype permet de partager des propriétés entre plusieurs objets, et il conduit naturellement à des notions d'héritage. Il permet aussi d'*augmenter* les objets pour leur rajouter des propriétés, bien après que ces objets aient été définis, y compris sur les types de base comme **String**.

Nous commencerons par voir un certain nombre de *patterns* fonctionnels, qui permettent de faire de la programmation objet avec des notions comme la visibilité, la structuration d'une application en modules (ou *packages*), des fabriques, ou encore des *patterns* permettant le découplage des composants d'une application à base d'événements, ou comme *subscriber/publisher*.

Ces *patterns* peuvent paraître déconcertant au premier abord pour un développeur habitué aux langages objet classiques. Avec un peu d'habitude, on en vient à considérer que *JavaScript* est un excellent langage objet, très expressif et très souple. Cependant, certains problèmes de conception du langage, qui n'ont pu être corrigés pour assurer la compatibilité ascendante, nécessitent quelques précautions, sous la forme de bonnes habitudes.

## 2.1 Passages d'Arguments, Objets **this** et *Pattern that*

### 2.1.1 Passage d'arguments par objets

En *JavaScript*, il est souvent plus pratique, plutôt que de passer une série de paramètres, ce qui oblige à tenir compte de l'ordre de ces paramètres, de donner en argument à une fonction les données dans les propriétés d'un objet, soit construit à la volée, soit construit auparavant.

Ce *pattern* offre souvent plus de souplesse que la manière classique. Dans l'exemple suivant, la fonction génère le code *HTML* de l'objet passé en paramètre, sans savoir de quel type d'objet il s'agit. On l'utilise ensuite pour afficher une adresse.

Code Source 2.1 : /pattern-fonct/ex02-affichageObjetBasic.js

```

1  /** @summary Crée un chaîne de caractère lisible qui représente l'objet.
2   * @description On suppose que toutes les propriétés de l'objet sont de type
3   * chaîne ou nombre (elles peuvent être automatiquement converties en chaîne)
4   * @function objectToHtmlTable
5   * @param {Object} spec - L'objet dont les propriétés doivent être affichées
6   * @return Le code HTML pour afficher les propriétés de l'objet
7   */
8  var objectToHtmlTable = function(spec){
9
10     var chaine = "<table><tbody>";
11     // Parcours des propriétés de l'objet spec passé en argument
12     for (propertyName in spec){
13         // La propriété est définie et non vide,
14         // elle ne vient pas du prototype de l'objet et ce n'est pas une fonction
15         if (spec[propertyName] && spec.hasOwnProperty(propertyName)
16             && typeof spec[propertyName] !== "function"){
17             // Concaténation à une chaîne. Les nombres sont convertis.
18             chaine += '<tr><td style="text-align: right;"><em>' + propertyName + " </em></td>' +
19                 "<td>" + spec[propertyName] + "</td></tr>";
20         }
21     };
22     chaine += "<tbody></table>";
23     return chaine;
24 };

```

Code Source 2.2 : /pattern-fonct/ex02-affichageObjetBasicTest.js

```

1  // Invocation de la fonction avec le pattern
2  var codeHTML = objectToHtmlTable({
3      id: "0f3ea759b1",
4      numeroRue: "2 bis",
5      rue: "Rue de la Paix",
6      complementAddr: "",
7      codePostal: "63000",
8      ville: "Clermont-Ferrand",
9      pays: "France"
10  });
11  // Utilisation de la valeur retournée pour générer la vue
12  document.getElementById("paragrapheResultat").innerHTML = codeHTML;

```

### 2.1.2 Invocation de Méthode avec le *Pattern* “*apply*”

Le *pattern apply* permet d’“appliquer” une méthode d’un certain objet à un autre objet. En d’autres termes, *pattern apply* permet d’exécuter une méthode d’un objet comme si elle était définie dans un autre objet. Plus précisément, lors de l’invocation de la méthode suivant le *pattern apply*, le code de la fonction est exécuté, mais chaque occurrence du mot réservé *this*, au lieu de faire référence à l’objet contenant la méthode, va faire référence à un autre objet, qui

est passé en premier argument lors de l'invocation. Les arguments ordinaires de la méthode, suivant la définition de ses paramètres, sont transmis, lors de l'invocation, dans un `Array` passé en second argument.

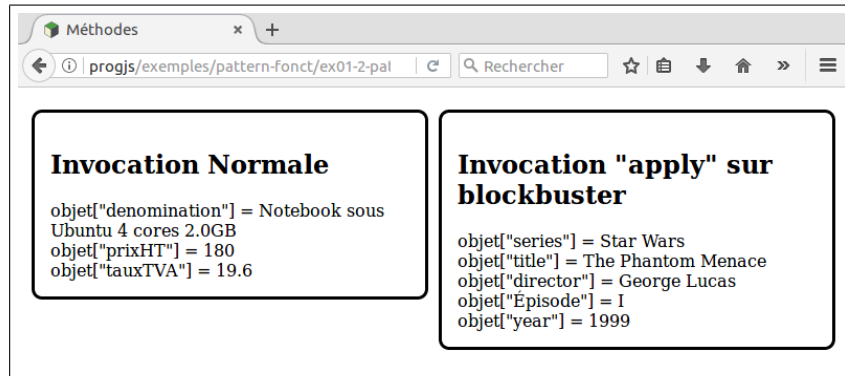


FIGURE 2.1 : Illustration du code source 2.3

Code Source 2.3 : `/pattern-fonct/ex01-2-patternApplyInvocationTest.js` (cf. Fig 2.1)

```

1  /** @description Littéral définissant un objet appelé "produit" */
2  var produit = {
3      "denomination" : "Notebook sous Ubuntu 4 cores 2.0GB",
4      "prixHT" : 180.0,
5      "tauxTVA" : 19.6,
6
7      /** @description Génère le code HTML pour afficher un objet quelconque
8       * @method getHtmlObjet
9       * @param {string} titre - texte du titre <h2>
10      * @param {string} balise - type de balise HTML contenant le code généré
11      *                               ("p", "em", "div", "span"... )
12      */
13     getHtml : function(titre, balise){
14         var chaine = "<" + balise + ">" // On ouvre la balise ;
15         chaine += "<h2>" + titre + "</h2>";
16         // Parcours de toutes les propriétés de l'objet (style "foreach") :
17         for (nom in this){
18             if (this.hasOwnProperty(nom) && typeof this[nom] !== "function"){
19                 chaine += "objet[" + nom + "] = "
20                     + this[nom] + "</br>"; // Appel de la méthode toString par défaut
21             }
22         }
23         chaine += "</" + balise + ">"; // On ferme la balise
24         return chaine;
25     }
26 };
27
28 /** @description Représente un film de la série "Star Wars" */
29 var blockbuster = {
30     "series" : "Star Wars",
31     "title" : "The Phantom Menace",
32     "director" : "George Lucas",
33     "Épisode" : "I",
34     "year" : "1999"

```

```

35 };
36
37 // Invocation classique de la méthode affichant le produit :
38 codeHTML = produit.getHtml("Invocation Normale", "span");
39 // Invocation avec pattern apply de la méthode de "produit"
40 // pour afficher "blockbuster" :
41 codeHTML += produit.getHtml.apply(blockbuster, // Objet prenant place de this
42                                   // Liste des arguments de la méthode :
43                                   ["Invocation \ "apply\ " sur blockbuster", // titre
44                                   "span"]); // balise
45 // Mise à jour de la vue
46 document.getElementById("paragrapheResultat").innerHTML = codeHTML;

```

### 2.1.3 Accès au Composite à partir du Composé (*pattern that*)

Une méthode d'un objet imbriqué, tel qu'un composé de la partie 1.4.3 obtenus par une fabrique, peut accéder aux propriétés du composite en utilisant une propriété ou une variable, traditionnellement appelée `that`, qui contient la référence du composite :

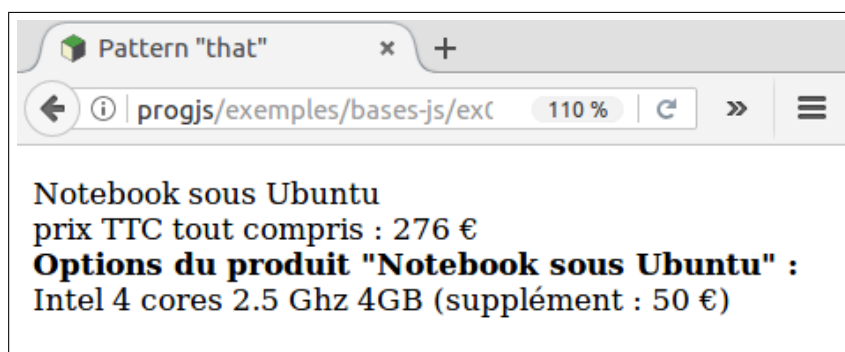


FIGURE 2.2 : Illustration du code source 2.4

Code Source 2.4 : `/pattern-fonct/ex06-returnedObjectCompositeAccess.js` (cf. Fig 2.2)

```

1  /** @description Littéral définissant un objet appelé "produit" */
2  var produit = {
3    denomination : "Notebook sous Ubuntu",
4    prixHT_base : 180.0,
5    tauxTVA : 20.0,
6    /** @description Construction d'un objet dans une méthode
7     */
8    getOptions : function() {
9      // Mémorisation du composite pour accès par le composé
10     var that = this; // pattern that (partie 1)
11     // Création de l'objet option dans la fonction ("fabrique")
12     return {
13       processor : "Intel 4 cores 2.5 Ghz",
14       memory : "4GB",
15       "prix supplémentaire HT" : 50.0,
16       /** @description Génère le code HTML des options
17        * @method getHTML
18        */
19       getHtml : function() {

```



```

20 // Accès au composite à partir d'une méthode du composé
21 var codeHTML = "<b>Options du produit \\"
22               + that.denomination // pattern that (partie 2)
23               + "\\\" :</b>";
24 codeHTML += "<br/>" + this.processor + " " + this.memory +
25            " (supplément : " + this["prix supplémentaire HT"] + " €euro;)";
26 return codeHTML;
27 }
28 };
29 },
30 /** @description Génère le code HTML complet du produit
31     * @method getHTML
32     */
33 getHtml : function(){
34     return this.denomination +
35            "<br/>prix TTC tout compris : "
36            + (this.prixHT_base
37              + (this.getOptions()["prix supplémentaire HT"] || 0.0))
38              *(1.0+this.tauxTVA/100.0)
39            + " €euro ;<br/>" + this.getOptions().getHtml() + "<br/>";
40 }
41 };
42
43 // appel d'une méthode :
44 codeHTML = "<p>" + produit.getHtml() + "</p>";
45 // Mise à jour de la vue
46 document.getElementById("paragrapheResultat").innerHTML = codeHTML;

```

## 2.2 Le *Pattern* Module

### 2.2.1 Cycle de Vie d'une Variable Locale à une Fonction

Une variable locale à une fonction, déclarée avec le mot clé **var**, n'est plus accessible à l'extérieur de la fonction. Cependant, contrairement à ce qui se passe dans un langage comme le *C*, la variable locale peut continuer à exister en mémoire après la fin de l'exécution de la fonction.

#### 2.2.1.a Rappel sur les variables locales en *C*

Dans le langage *C*, les variables locales à une fonction sont créées dans une zone mémoire appelée la *pile*, et sont automatiquement détruites lors du retour de la fonction après son invocation.

Dans l'exemple suivant, une fonction *C* retourne l'adresse d'une variable locale, puis la fonction **main** tente d'accéder à la zone mémoire correspondante.

Code Source 2.5 : /pattern-fonct/fonction-C/variableLocale.c

```

1 #include <stdio.h>
2
3 /** Fonction buggée qui retourne l'adresse d'une variable locale */
4 int* maFonction(){
5     int x = 2; // variable locale x
6     int* pRetour = &x; // Pointeur sur la variable x
7     printf("%d\n", *pRetour); // Affiche x, donc 2

```

```

8   return pRetour; // Retourne l'adresse d'une variable locale !!!!!
9 }
10
11 int main() {
12     int *pointeurSurX = maFonction(); // Invocation de la fonction
13     printf("%d\n", *pointeurSurX); // Erreur mémoire !!!!
14     return 0;
15 }

```

Même si l'exécution peut éventuellement fonctionner sur certains compilateurs, le code n'est pas défini dans le standard du *C ANSI*, et le comportement peut donner n'importe quoi ! On peut mettre en évidence le *bug* avec l'outil *valgrind*, qui détecte une erreur mémoire (ici sur la ligne 13) :

```

==3894== Use of uninitialised value of size 8
==3894==    at 0x4E8476B: _itoa_word (_itoa.c:179)
==3894==    by 0x4E8812C: vfprintf (vfprintf.c:1631)
==3894==    by 0x4E8F898: printf (printf.c:33)
==3894==    by 0x400619: main (variableLocale.c:13)

```

### 2.2.1.b Survivance des variables locales en *JavaScript*

En *JavaScript*, le cycle de vie d'une variable locale peut aller au delà du retour de la fonction après son invocation, si on retourne une entité (objet, fonction, tableau, etc.) qui fait référence à cette variable locale. Ceci est possible du fait que les variables locales ne sont pas gérées en mémoire via une pile, mais comme de la mémoire dynamique libérée par le *garbage collector*. La mémoire pour la variable locale ne peut donc pas être détruite tant qu'il existe une référence à cette variable !

Dans l'exemple suivant, une fonction `maFonction` retourne une autre fonction locale `accesVariableX`, qui elle-même retourne la variable locale `x` de `maFonction`. On peut alors accéder, via la fonction locale retournée, à la variable locale `x`, bien après que l'exécution de la fonction `maFonction`



FIGURE 2.3 : Illustration du code source 2.6

Code Source 2.6 : `/pattern-fonct/ex01-survieVariableLocaleJS.js` (cf. Fig 2.3)

```

1
2 var maFonction = function() {
3     var x = 2; // Variable locale x de type "number", valeur égale à 2
4
5     // Fonction locale qui accède à la variable locale x et la retourne
6     var accesVariableX = function() {
7         return x; // Utilisation de la variable locale x
8     };
9

```

```

10   return accesVariableX ; // On retourne la fonction locale
11 };
12
13 var getX = maFonction() ; // On récupère la fonction retournée par maFonction
14 codeHTML = "La variable locale x a pour valeur " + getX() ;
15 document.getElementById( "paragrapheResultat" ).innerHTML = codeHTML ;

```

## 2.2.2 Principe Général du *Pattern* Module

Le *pattern Module* permet de créer des composants qui peuvent jouer le rôle que jouent les classes dans les langages objet classiques. Il permet, entre autre, de créer des données et méthodes privées, et une interface publique avec d'autres données et méthodes, qui sont accessibles de l'extérieur, et qui peuvent, elles, accéder aux propriétés privées.

Le *pattern* consiste à créer une fonction. Les données et méthodes privées sont simplement des variables locales de la fonction. Elles ne sont donc pas visibles du monde extérieur à la fonction. La fonction renvoie un objet, qui constitue l'interface publique du module, dont les propriétés (données, objets ou fonctions) accèdent aux variables privées. Lorsque l'objet est retourné, on ne peut plus accéder directement aux variables locales de la fonction, mais celles-ci restent vivantes (leur cycle de vie ne se termine pas) tant que l'objet retourné qui s'y réfère n'est pas lui-même détruit. Autrement dit, on peut continuer à manipuler ces variables locales au travers des méthodes de l'interface publique.

Code Source 2.7 : /pattern-fonct/ex01-modulePattern.js

```

1  /** @description Illustration du pattern Module
2   * permettant d'implémenter la visibilité en JavaScript.
3   */
4  var mySecretModule = function(defaultSecretValue){
5    /** @description Donnée privée avec une valeur par défaut
6     *      (variable locale)
7     * @private {string} PrivateSecret = (defaultSecretValue | "")
8     */
9    var myPrivateSecret = ((defaultSecretValue && // s'il y a une valeur en
        paramètre
10                               typeof defaultSecretValue === "string")
11                               && defaultSecretValue)
12                               || ""; // si pas de valeur en paramètre
13
14    /** @description Teste une expression régulière (fixée) sur une chaîne
15     * @function myRegexTestMethod
16     * @private
17     * @param {string} chaine - La chaîne à tester
18     */
19    var myRegexTestMethod = function(chaine){
20      return (typeof chaine === "string") && /^[a-z]*$/i.test(chaine) ;
21    };
22
23    /** @summary Interface publique du module, suivant le "pattern module".
24     * @description On crée un objet qui va être rendu public
25     * Cet objet va être retourné, mais pas les données privées.
26     * Les méthodes de cet objet constitueront les méthodes publiques du module,
27     * qui utilisent les variables (et fonctions) privées (variables locales).
28     */
29    var publicInterface = {

```

```

30  /** @description Donnée publique avec une valeur par défaut
31  * @public {string} donneePublique = 'donnée publique par défaut'
32  */
33  donneePublique : 'donnée publique par défaut',
34
35  /** @description Setter pour modifier la donnée privée myPrivateSecret
36  * @method setSecret
37  * @public
38  * @param {string} secretValue - La valeur à tester puis affecter
39  * @throws Si secretValue ne suit pas l'expression régulière
40  *         myRegexTestMethod
41  */
42  setSecret : function(secretValue){
43      // Test d'expression régulière via la fonction privée :
44      if (myRegexTestMethod(secretValue)){
45          myPrivateSecret = secretValue;
46      }else{
47          throw {
48              name: "IllegalArgumentException",
49              message: "Le secret " + secretValue + " est invalide."
50          };
51      }
52  },
53
54  /** @description Accesseur pour accéder à la donnée privée myPrivateSecret
55  * @method getSecret
56  * @public
57  * @return La valeur de la donnée privée myPrivateSecret
58  */
59  getSecret : function(){
60      return myPrivateSecret;
61  },
62  }; // Fin de publicInterface
63  // L'interface publique du module est retournée pour utilisation hors de la
64  // fonction
65  return publicInterface;
66  }

```

Le fichier suivant teste les accès aux données et méthodes publiques du module suivant le *pattern module* :

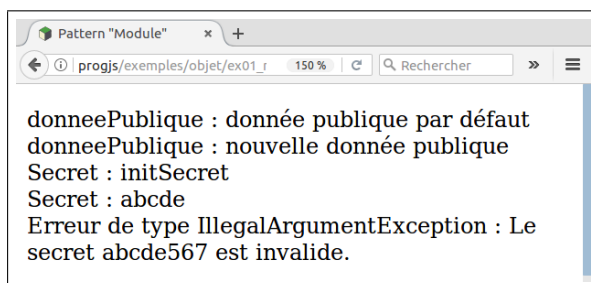


FIGURE 2.4 : Illustration du code source 2.8

Code Source 2.8 : `/pattern-fonct/ex01-modulePatternTest.js` (cf. Fig 2.4)

```

1  // Création du module par invocation du "pattern module" :

```

```

2 var secretModule = mySecretModule("initSecret");
3 // Code HTML à ajouter au paragraphe d'ID paragrapheResultat
4 var codeHTML = "";
5 // Accès à la donnée publique de l'interface :
6 codeHTML += "donneePublique : " + secretModule.donneePublique + "<br/>";
7 // Modification de la donnée publique de l'interface :
8 secretModule.donneePublique = "nouvelle donnée publique";
9 codeHTML += "donneePublique : " + secretModule.donneePublique + "<br/>";
10
11 // Accesseur de secret (variable locale privée du "pattern module") :
12 codeHTML += "Secret : " + secretModule.getSecret() + "<br/>";
13 // Tentative de modifier le secret (le setter public fait des tests) :
14 try{
15     secretModule.setSecret("abcde"); // La chaîne n'entraîne pas d'erreur
16     codeHTML += "Secret : " + secretModule.getSecret() + "<br/>";
17 } catch (e) {
18     codeHTML += "Erreur de type " + e.name + "<br/>Message : " + e.message + "<br/>";
19 }
20 // Tentative de modifier le secret (le setter public fait des tests) :
21 try{
22     secretModule.setSecret("abcde567"); // La chaîne génère une erreur
23     codeHTML += "Secret : " + secretModule.getSecret() + "<br/>";
24 } catch (e) {
25     codeHTML += "Erreur de type " + e.name + " : " + e.message + "<br/>";
26 }
27 document.getElementById("paragrapheResultat").innerHTML = codeHTML;

```

Le mécanisme du langage essentiel pour ce *pattern* est la portée (*scope*) des variables locales à une fonction, qui s'étend aux sous-fonctions de la fonction, et à leurs sous-fonctions...

## 2.3 Exemple de fabrique sommaire

Dans l'exemple suivant, une fabrique, suivant un *pattern* module très sommaire, construit un objet de type adresse (éventuellement partiellement rempli), en sélectionnant les propriétés que l'objet en paramètre qui sont dans une liste.

Cet exemple est plutôt un *exemple d'école* et nous verrons plus loin des exemple plus complets, les propriété de l'objet retourné manipulant des données (attributs) privées.

Code Source 2.9 : /pattern-fonct/ex03-methodLiteralParam.js

```

1 /** @description Fabrique créant un objet de type "Adresse"
2  * avec des "attributs" (des propriétés attendues déterminées) fixés.
3  * On vérifie que les propriétés sont bien spécifiées pour les "attributs".
4  * @param {Object} spec - Objet avec les données d'une adresse
5  * @param {string} spec.id - l'identifiant unique de l'adresse
6  * @param {string} spec.numeroRue - le numéro de la rue/place
7  * @param {string} spec.rue - la rue/place
8  * @param {string} spec.complementAddr - Le bâtiment, lieu dit...
9  * @param {string} spec.codePostal - Le code postal
10 * @param {string} spec.ville - La ville
11 * @param {string} spec.pays - Le pays
12 * @return Une instance d'adresse "validée" (objet avec les propriétés)
13 * @throws Si "spec" ne contient pas toutes les propriétés attendues.
14 */

```

```

15 var fabriqueAdresseVersion1 = function(spec){
16   // Objet à retourner initialement vide
17   var adresse = {};
18   // Liste des "attributs" d'une adresse (propriétés attendues)
19   var listeProprietes = [ "id", "numeroRue", "rue", "complementAddr",
20                           "codePostal", "ville", "pays" ];
21   // Parcours des propriétés de l'objet spec passé en argument
22   for (propertyName in spec){
23     if (spec.hasOwnProperty(propertyName)){
24       // Si la propriété existe dans le type adresse :
25       if (listeProprietes.indexOf(propertyName) >= 0){
26         adresse[propertyName] = spec[propertyName];
27       }else{
28         throw {
29           name : "UnknownPropertyException",
30           message : "Propriété de l'adresse inconnue."
31         };
32       }
33     }
34   }
35   return adresse;
36 };

```

Code Source 2.10 : /pattern-fonct/ex03-methodLitteralParamTest.js

```

1  // création d'une instance avec la fabrique
2  var adresse = fabriqueAdresseVersion1({
3    id : "0f3ea759b1",
4    numeroRue : "2 bis",
5    rue : "Rue de la Paix",
6    complementAddr : "",
7    codePostal : "63000",
8    ville : "Clermont-Ferrand",
9    pays : "France"
10 });
11 // Génération d'HTML par introspection :
12 codeHTML = objectToHtmlTable(adresse);
13 // Utilisation de la valeur retournée pour générer la vue
14 document.getElementById("paragrapheResultat").innerHTML = codeHTML;

```

## 2.4 Structuration d'une application

L'un des principaux défauts de *JavaScript* est sa tendance à créer, parfois sans faire exprès, des variables globales, ce qui a tendance à créer des interactions involontaires entre des parties du code qui n'ont rien à voir, ce qui génère des *bugs* difficiles à débuser...

Nous allons voir maintenant comment rédiore les nombres de variables globales de notre programme à une seule variable, ici appelée **myApp**, qui contient toute notre application.

L'objet **myApp**, initialement, ne contient que deux méthodes :

- Une méthode **addModule** qui permet d'ajouter un objet quelconque (de type **pattern-fonct**, **Function**, **Array**, etc.) sous la forme de propriété de l'application.

- Une méthode `init`, qui permet de rajouter un ensemble de propriétés prédéfinies, sans avoir à les créer une par une.

Code Source 2.11 : `/pattern-fonct/ex04-structureApplication.js`

```

1  /** Définition d'une variable application.
2  * L'application est initialement vide et ne comporte que la fonctionnalité
3  * permettant d'ajouter des modules.
4  * Une méthode init() permet d'initialiser plusieurs modules.
5  *
6  */
7  var myApp = {
8    /** Méthode qui ajoute un module à notre application
9    * Un module peut être n'importe quel objet qui contient
10   * des données ou des méthodes...
11   * @method addModule
12   * @param {Object/function/string/regex/number /Array/...} moduleObject
13   *          - un objet ou valeur quelconque à ajouter à notre application.
14   */
15   addModule : function(moduleName, moduleObject){
16     if (typeof moduleName === "string" &&
17         /^[a-z]{1,}[a-z0-9\_]*/i.test(moduleName)){
18       this[moduleName] = moduleObject;
19     }else{
20       throw {
21         name : "IllegalArgumentException",
22         message : "Impossible de créer les module : nom " + moduleName
23                 + " illégal"
24       }
25     }
26   },
27
28   /** Ajoute toutes les propriétés d'un objet à notre application.
29   * @method init
30   * @param {Object} spec - objet contenant les propriétés à ajouter.
31   */
32   init : function(spec){
33     for (var attributeName in spec){
34       if (spec.hasOwnProperty(attributeName)){
35         this.addModule(attributeName, spec[attributeName]);
36       }
37     }
38   }
39 };
40
41 // Initialisation de l'application avec un module metier initialement vide.
42 myApp.init({
43   metier : {}
44 });

```

Nous utilisons maintenant ce squelette d'application et nous créons dans notre application un module `metier`.

Nous utilisons ensuite le *pattern apply* qui nous permet d'utiliser la méthode `myApp.addModule` en prenant comme `"this"` un autre objet que `myApp`.

En appliquant donc la méthode `myApp.addModule` en prenant `myApp.metier` comme `"this"`,

nous créons un sous-module de `myApp.metier`, appelé `myApp.metier.sousModule`. Ce sous-module contient une propriété `essai`.

Code Source 2.12 : `/pattern-fonct/ex04-structureApplicationTest.js`

```

1 // ajout d'une propriété au métier :
2 myApp.metier.coucou = "test";
3
4 // Ajout d'un sous-module au module myApp.metier
5 // On applique (pattern apply) addModule en prenant this = myApp.metier
6 myApp.addModule.apply(myApp.metier,
7     [ "sousModule",
8       {essai: "Je suis la propriété \"essai\" du sous module"}
9     ]
10 );
11 /** @description Module myApp.view avec une propriété "codeHTML"
12  * @module myApp.view
13  * La propriété "codeHTML" contiendra le code HTML de la vue
14  * (inséré dans le paragraphe d'ID "paragrapheResultat")
15  */
16 myApp.addModule("view", {codeHTML: ""});
17
18 /** @description Programme Principal de l'application
19  * @method mainFunction
20  * @public
21  */
22 myApp.addModule("mainFunction", function(){
23     myApp.view.codeHTML += "Exécution de myApp.mainFunction...<br/>";
24     myApp.view.codeHTML += "myApp.metier.coucou : "
25         + myApp.metier.coucou + "<br/>";
26     myApp.view.codeHTML += "myApp.metier.sousModule.essai : "
27         + myApp.metier.sousModule.essai;
28 });
29
30 // Exécution de la méthode mainFunction
31 myApp.mainFunction();
32 // Utilisation de "myApp.view.codeHTML" pour générer la vue
33 document.getElementById("paragrapheResultat").innerHTML
34     = myApp.view.codeHTML;

```

”

Code Source 2.13 : Fichiers *JS* inclus dans `ex04-structureApplicationTest.html`

```

1 <p id="paragrapheResultat"></p>
2 <script src="ex04-structureApplication.js"></script>
3 <script src="ex04-structureApplicationTest.js"></script>

```

## 2.5 Exemple : un module `metier.regexUtil`

L'exemple suivant montre l'utilisation du *pattern* Module pour créer un sous-module métier utilitaire pour tester des expressions régulières courantes :

- Expressions formées avec les caractères du langage courant dans une des langues dont les accents sont normalisés dans la norme ISO 8859 – 1 (*Latin-1*, Europe occidentale), admettant aussi les guillemets, apostrophes et traits d'union (tiret haut).



- Mêmes caractère que la précédente mais admettant en outre les chiffres.
- Mêmes caractère que la précédente mais admettant en outre les caractères de ponctuation (; . , ! ? :) et les parenthèses.

Trois expressions régulières constantes (donc pré-compilées) sont définies comme données statiques (en un seul exemplaire) privées. L'interface fournit trois méthodes pour tester ces expressions régulières sur une chaîne, avec éventuellement des conditions de longueur minimale ou maximale sur la chaîne (exemple : champs obligatoire...).

Code Source 2.14 : /pattern-fonct/ex05-modulePatternRegex.js

```

1  /** @description Sous-module du métier : utilitaire pour expressions régulières
2   * L'objet créé est l'interface publique retournée par une fonction
3   * qui suit le "pattern module".
4   * La fonction retourne son interface : un objet avec ses méthodes.
5   * Cet objet est ajouté comme sous-module au module "metier" par "addModule".
6   * @module regexUtil
7   * @augments myApp.metier
8   */
9  myApp.addModule.apply(myApp.metier, [ "regexUtil", function() {
10
11      //////////////////////////////////////
12      // Variables et méthodes "statiques" privées
13
14      /** @description Expression régulière constante pour la langue naturelle
15       * Admet les caractères d'europe occidentale (jeu latin 1) et espaces.
16       * @constant
17       * @private
18       */
19      var regexLatin1
20      = /^[a-zA-ZÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖ×ØÙÚÛÜÝÞßàáâãäåæ,èéêëìíîïðñóôõ÷øùúû
21      Ääÿþÿ\s"'\-]*$/i;
22
23      /** @description Expression régulière constante pour la langue et chiffres
24       * @constant
25       * @private
26       */
27      var regexLatin1WithDigits =
28      /^[a-zA-ZÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖ×ØÙÚÛÜÝÞßàáâãäåæ,èéêëìíîïðñóôõ÷øùúû
29      Ääÿþÿ\s"'\-0-9]*$/i;
30
31      /** @description Expression regex constante langue + chiffres + ponctuation
32       * @constant
33       * @private
34       */
35      var regexLatin1WithDigitsPunctuation =
36      /^[a-zA-ZÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖ×ØÙÚÛÜÝÞßàáâãäåæ,èéêëìíîïðñóôõ÷øùúû
37      Ääÿþÿ\s"'\-0-9|;|.|!|?|:|(|)]*/i;
38
39      /** @description Valide une expression régulière sur une chaîne
40       * avec conditions de longueur (minimale ou maximale) optionnelles
41       * @method validateRegex
42       * @private
43       * @param {Object} spec - objet contenant les données du test à effectuer
44       * @param {function} spec.regexTest -

```

```

42      *                               fonction de test qui renvoie true en cas de succès
43      *                               et un message d'erreur en cas d'échec
44      * @param {string} spec.chaine – chaîne de caractères à tester
45      * @param {number} [spec.minLength=0] – longueur minimale pour la chaîne
46      * @param {number} [spec.maxLength=undefined] – longueur maximale pour la
      chaine
47      * @return {boolean|string} true si les conditions sont satisfaites ,
48      *                               un message d'erreur pour un utilisateur sinon.
49      */
50      var validateRegex = function(spec){
51          if (typeof spec.chaine === "string"
52              && (!spec.minLength || spec.chaine.length >= spec.minLength)
53              && (spec.maxLength === undefined ||
54                  spec.chaine.length <= spec.maxLength)
55              ){
56              return spec.regex.test(spec.chaine);
57          }
58          return "Erreur : longueur de l'entrée (champ obligatoire , trop long...)";
59      };
60
61      /** @description Interface publique du "pattern module" pour regexUtils ,
62      * retourné par la fonction , contient les méthodes publiques du module. */
63      var publicInterfaceRegex = {
64          /** @description teste l'expression du langage naturel avec espaces
65          * @method testRegexLatin1
66          * @public
67          * @param {Object} spec – objet contenant les données du test à effectuer
68          * @param {string} spec.chaine – chaîne de caractères à tester
69          * @param {number} [spec.minLength=0] – longueur minimale de la chaîne
70          * @param {number} [spec.maxLength=undefined] – longueur max de la chaîne
71          * @return {boolean|string} true si les conditions sont satisfaites ,
72          *                               un message d'erreur pour un utilisateur sinon.
73          */
74          testRegexLatin1 : function(spec){
75              // Ajout d'une propriété à spec (augmentation)
76              spec.regex = regexLatin1;
77              return validateRegex(spec);
78          },
79
80          /** @description teste l'expression du langage naturel , espaces , chiffres
81          * @method testRegexLatin1WithDigits
82          * @public
83          * @param {Object} spec – objet contenant les données du test à effectuer
84          * @param {string} spec.chaine – chaîne de caractères à tester
85          * @param {number} [spec.minLength=0] – longueur minimale de la chaîne
86          * @param {number} [spec.maxLength=undefined] – longueur max de la chaîne
87          * @return {boolean|string} true si les conditions sont satisfaites ,
88          *                               un message d'erreur pour un utilisateur sinon.
89          */
90          testRegexLatin1WithDigits : function(spec){
91              // Ajout d'une propriété à spec (augmentation)
92              spec.regex = regexLatin1WithDigits;
93              return validateRegex(spec);
94          },
95
96          /** @description teste le langage naturel , espaces , chiffres et

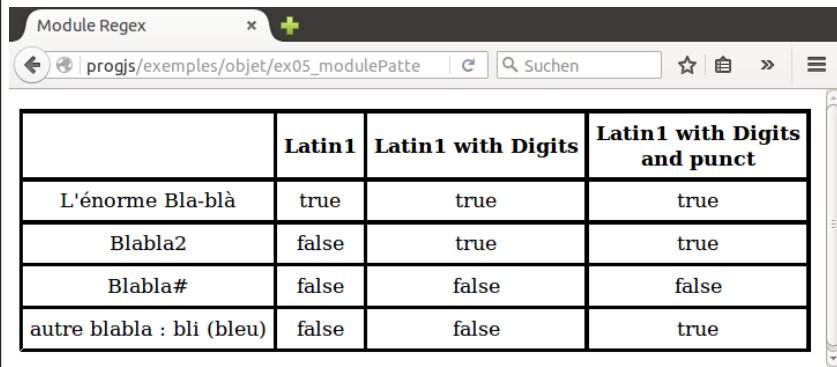
```

```

    punctuation
97  * @method testRegexLatin1WithDigitsPunctuation
98  * @public
99  * @param {Object} spec - objet contenant les données du test à effectuer
100 * @param {string} spec.chaine - chaîne de caractères à tester
101 * @param {number} [spec.minLength=0] - longueur minimale de la chaîne
102 * @param {number} [spec.maxLength=undefined] - longueur max de la chaîne
103 * @return {boolean/string} true si les conditions sont satisfaites,
104 *                               un message d'erreur pour un utilisateur sinon.
105 */
106 testRegexLatin1WithDigitsPunctuation : function(spec){
107     // Ajout d'une propriété à spec (augmentation)
108     spec.regex = regexLatin1WithDigitsPunctuation;
109     return validateRegex(spec);
110 }
111 }; // fin du littéral définissant publicInterfaceRegex
112
113 // On retourne l'objet contenant l'interface publique (pattern "module").
114 return publicInterfaceRegex;
115
116 }()) // fin ET APPEL de la fonction qui crée l'objet "regexUtil"
117 ); // fin de l'appel de la méthode myApp.addModule avec le pattern "apply"
118 // (ajout de l'objet publicInterfaceRegex au metier, sous le nom regexUtil)

```

Le fichier *HTML* réalise des tests des méthodes du module `regexUtil` sur un jeu de chaînes, et affiche les résultats dans une table.



The screenshot shows a web browser window with the address bar displaying 'progjs/exemples/objet/ex05\_modulePatte'. The page content is a table with 4 columns: an empty header cell, 'Latin1', 'Latin1 with Digits', and 'Latin1 with Digits and punct'. There are 4 rows of data.

|                           | Latin1 | Latin1 with Digits | Latin1 with Digits and punct |
|---------------------------|--------|--------------------|------------------------------|
| L'énorme Bla-blà          | true   | true               | true                         |
| Blabla2                   | false  | true               | true                         |
| Blabla#                   | false  | false              | false                        |
| autre blabla : bli (bleu) | false  | false              | true                         |

FIGURE 2.5 : Illustration du code source 2.15

Code Source 2.15 : `/pattern-fonct/ex05-modulePatternRegexTest.js` (cf. Fig 2.5)

```

1  /** @description Programme Principal : test du module myApp.metier.regexUtil
2  * @method mainFunction
3  * @public
4  */
5  myApp.addModule("mainFunction", function(){
6      // Chaînes pour les tests d'expressions régulières :
7      var tabChaines = ["L'énorme Bla-blà", "Blabla2", "Blabla#", "autre blabla :
          bli (bleu)"];
8      var i;
9      // Raccourci par copie de référence :
10     var regexUtil = myApp.metier.regexUtil;
11     var codeHTML = "<table><thead>" +

```

```

12     "<tr><th></th><th>Latin1</th><th>Latin1 with Digits</th>" +
13     "<th>Latin1 with Digits<br />and punct</th></tr></thead><tbody>";
14     for (i = 0 ; i < tabChaines.length ; i++){
15         codeHTML += "<tr><td>" + tabChaines[i] + "</td>"
16             + "<td>" + regexUtil.testRegexLatin1({
17                 chaine : tabChaines[i]
18             }) + "</td>"
19             + "<td>" + regexUtil.testRegexLatin1WithDigits({
20                 chaine : tabChaines[i]
21             }) + "</td>"
22             + "<td>" + regexUtil.testRegexLatin1WithDigitsPunctuation({
23                 chaine : tabChaines[i]
24             }) + "</td>"
25     }
26     codeHTML += "</tbody></table>";
27     // Utilisation de "myApp.view.codeHTML" pour générer la vue
28     document.getElementById("paragrapheResultat").innerHTML = codeHTML;
29 } ; // fin de myApp.mainFunction()
30
31 // Lancement de l'application :
32 myApp.mainFunction() ;

```

Code Source 2.16 : Fichiers *JS* inclus dans ex05-modulePatternRegexTest.html

```

1 <!-- Création de l'application vide avec deux méthodes -->
2 <script src="ex04_structureApplication.js"></script>
3 <!-- Création de sous-module regexUtil de myApp.metier -->
4 <script src="./ex05-modulePatternRegex.js"></script>
5 <!-- Création de la méthode de test myApp.metier.mainFunction -->
6 <script src="./ex05-modulePatternRegexTest.js"></script>

```

## 2.6 Modélisation de Modules Métier (version 1)

Nous voyons maintenant un utilitaire général permettant d'ajouter dans `myApp.metier` un module pour représenter ce qu'on appellerait dans les langages objets classiques une *classe métier*. Nous utiliserons dans la suite de ce cours les conventions et le vocabulaire suivant :

- Un module métier permet de représenter des entités relevant du métier qui possèdent (regroupent) des attributs.  
**Exemple.** Un module métier `myApp.metier.adresse` regroupera les attributs `id`, `numeroRue`, `rue`, `complementAddr`, `codePostal`, `ville` et `pays`...
- Chaque module métier comprendra des *méthodes d'instances*, qui s'appliqueront aux instances, et dont l'implémentation s'appuiera sur les valeurs des attributs.  
**Exemple.** La méthode d'instance `getAttribute`, appelée *accesseur*, permet d'obtenir la valeur d'un attribut dans une instance à partir du nom de l'attribut.
- Chaque module métier comprendra des *méthodes de classe*, dont l'implémentation et le résultat seront indépendants des valeurs des attributs, donc indépendante des instances.  
**Exemple.** La méthode de classe `getAttributeList` renvoie, sous forme d'`Array`, la liste des noms d'attributs du module métier (liste commune à toutes les instances d'un même module métier).

Chaque module métier est construit à partir d'un objet `attributesPatterns`, qui définit la structure d'une instance. Ici, `attributesPatterns` définit, pour chaque attribut, une méthode de test par expression régulière de validité d'une valeur pour l'attribut, et une propriété `labelText` à afficher pour indiquer à l'utilisateur de quelle donnée il s'agit (typiquement : texte de l'élément *HTML* `<label>` associé à un `input` dans un formulaire). On pourrait facilement adapter le code pour permettre des propriétés calculées, ou encore différents types d'éléments d'interface utilisateur (divers *inputs* (couleur, nombre, date,...), de *textarea* ou de *select*) dans les formulaires, etc.

### 2.6.1 Attributs et méthodes statiques (Version 1)

L'interface publique du module métier propose quelques méthodes statiques utilitaires, comme l'accès à la liste des noms d'attributs, aux données `labelText`, ou le test d'expression régulière d'une valeur pour un attribut.

Code Source 2.17 : `/pattern-fonct/ex06-createModuleMetier.js`

```

1  /** @description Définit les propriétés générale des objets métiers
2   * à partir d'une spécification des attributs.
3   * On ajoute au métier un objet qui est l'interface publique d'une fonction qui
4   * suit le pattern "module".
5   * La fonction retourne son interface publique qui est un objet.
6   * Cet objet est ajouté comme sous-module au module "metier".
7   *
8   * Dans cet objet, on ne trouve pas pour le moment les propriétés d'instance.
9   * Celles-ci seront ajoutées par "augmentation".
10  *
11  * @module createModuleMetier
12  * @augments myApp.metier
13  * @param {string} moduleName - nom du module métier
14  * @param {Object} attributesPatterns - objet contenant les attributs d'un
15  *   module métier
16  *   - Chaque propriété de attributesPatterns définit un attribut
17  *   du module métier
18  *   - chaque attribut possède
19  *   + une fonction de test sur les valeurs nommée regexTest
20  *   + un texte de label permettant de désigner la donnée
21  *   pour l'utilisateur
22  */
23  myApp.addModule.apply(myApp.metier, [ "createModuleMetier",
24                                     function(moduleName, attributesPatterns){
25
26     //////////////////////////////////////
27     // Propriétés et méthodes "statiques" privées
28
29     /**
30      * Tableau contenant la liste des attributs d'une instance.
31      * Le tableau est précalculé lors de l'initialisation.
32      * @member
33      * @private
34      */
35     var attributeList = function(){
36         var liste = [];
37
38         // Parcours des propriétés de l'objet attributesPatterns.regexTest

```

```

34 // qui correspondent aux attributs de l'instance
35 for (var attributeName in attributesPatterns){
36 // Ne pas considérer les propriétés "héritées" du prototype.
37 if (attributesPatterns.hasOwnProperty(attributeName)){
38 liste.push(attributeName);
39 }
40 }
41
42 return liste;
43 }(); // appel immédiat de la fonction anonyme.
44
45
46 //////////////////////////////////////
47 // Interface publique du module
48
49 /** @description Objet contenant les données et méthodes publiques
50 * (les propriétés publiques sont retournées par la fonction "module").
51 */
52 var publicInterfaceModulesMetier = {
53
54 /** @description Renvoie la liste des noms d'attributss des instances.
55 * @method getAttributeList
56 */
57 getAttributeList : function(){
58 return attributeList;
59 },
60
61 /** @description Renvoie le texte de description d'un attribut des instances
62 *
63 * @method getLabelText
64 * @param {string} attributeName – nom de propriété
65 * @return {string} le texte de description courte du champs
66 */
67 getLabelText : function(attributeName){
68 return attributesPatterns[attributeName].labelText;
69 },
70
71 /** @description Expose le test d'expression régulière des attributs
72 * des instances.
73 * Peut être utilisée pour le filtrage des données d'un formulaire.
74 * @method testRegex
75 * @param {string} attributeName – nom de propriété
76 * @param {string} value – valeur pour initialiser l'attribut
77 * @return {boolean/string} true si la chaîne est un code postal valide,
78 * un message d'erreur sinon.
79 */
80 testRegex : function(attributeName, value){
81 if (attributesPatterns[attributeName] === undefined){
82 return "La propriété " + attributeName + " n'existe pas";
83 }else{
84 return attributesPatterns[attributeName].regexTest(value);
85 }
86 }
87 }; // fin de l'objet publicInterfaceModulesMetier
88

```

```

89 myApp.metier[moduleName] = publicInterfaceModulesMetier ;
90
91 } // fin de la fonction qui crée l'objet myApp.metier[moduleName]
92
93 ); // fin de l'appel "apply" de la méthode myApp.addModule
94 // (ajout de l'objet publicInterfaceModulesMetier au metier, sous le nom
    createModuleMetier)

```

## 2.6.2 Exemple : Module Métier adresse

Nous créons (nousinstancions) maintenant un sous-module `myApp.metier.adresse`.

Code Source 2.18 : `/pattern-fonct/ex06-moduleMetierAdresse.js`

```

1  /** @description Module métier myApp.metier.adresse
2   * permettant de représenter des adresses postales.
3   *
4   * @module
5   * @public
6   *
7   * @property {function(string)} getAttribute - permet d'obtenir la valeur d'un
    attribut à partir du nom de l'attribut
8   * @property {function(string)} getLabelText - permet d'obtenir le texte de
    label d'un attribut à partir du nom de l'attribut
9   * @property {function(string, value)} testRegex - permet de tester la validité
    d'une valeur pour un attribut
10  */
11 myApp.metier.createModuleMetier("adresse",
12 /** @description Spécifie les attributs des objets de type adresse :
13  * propriétés attendues, forme de ces données...
14  *
15  * @private
16  *
17  * @property {Object} id - Propriétés de l'identifiant unique de l'instance
18  * @property {Object} numéroRue - Propriétés du numéro de la rue
19  * @property {Object} rue - Propriétés du nom de la rue/place
20  * @property {Object} complementAddr - Propriétés du complément Lieu dit/Bâ
    timent...
21  * @property {Object} codePostal - Propriétés du code postal
22  * @property {Object} ville - Propriétés du nom de la ville
23  * @property {Object} numéroRue - Propriétés du nom du pays
24  */
25 {
26     id : {
27         regexTest : function(chaine){
28             if (/^[0-9a-f]{10}$/i.test(chaine) === true){
29                 return true;
30             }else{
31                 return "L'identifiant doit comporter 10 chiffres hexa.";
32             }
33         },
34         labelText : "Identifiant"
35     },
36     numeroRue : {
37         regexTest : function(chaine){
38             if (myApp.metier.regexUtil.testRegexLatin1WithDigits({

```

```

39         chaine : chaine ,
40         maxLength : 15
41     }) == true)
42     {
43         return true ;
44     }else{
45         return "Le numéro de la rue contient au plus 15 caractères , "
46             + "lettres , tirets et guillemets ou chiffres." ;
47     }
48 },
49 labelText : "Numéro "
50 },
51 rue : {
52     regexTest : function(chaine){
53         if (myApp.metier.regexUtil.testRegexLatin1WithDigits({
54             chaine : chaine ,
55             minLength : 1,
56             maxLength : 255
57         }) == true)
58         {
59             return true ;
60         }else{
61             return "le nom de la rue/place , obligatoire ne contient que "
62                 + " des lettres , tirets et guillemets ou chiffres." ;
63         }
64     },
65     labelText : "rue/place "
66 },
67 complementAddr : {
68     regexTest : function(chaine){
69         if (myApp.metier.regexUtil.testRegexLatin1WithDigitsPunctuation({
70             chaine : chaine ,
71             maxLength : 255
72         }) == true)
73         {
74             return true ;
75         }else{
76             return "le complément d'adresse ne contient que des lettres , "
77                 + "tirets et guillemets ou chiffres." ;
78         }
79     },
80     labelText : "Lieu dit , Bâtiment , BP"
81 },
82 codePostal : {
83     regexTest : function(chaine){
84         if (/^[0-9]{5}$/.test(chaine) == true){
85             return true ;
86         }else{
87             return "Le code postal doit comporter 5 chiffres décimaux." ;
88         }
89     },
90     labelText : "Code Postal "
91 },
92 ville : {
93     regexTest : function(chaine){
94         if (myApp.metier.regexUtil.testRegexLatin1({

```



```

95         chaine : chaine ,
96         minLength : 1,
97         maxLength : 255
98     }) == true)
99     {
100         return true ;
101     } else {
102         return "La ville , obligatoire , ne contient que des lettres , "
103             + "tirets et guillemets ." ;
104     }
105 },
106 labelText : "Ville "
107 },
108 pays : {
109     regexTest : function(chaine){
110         if (myApp.metier.regexUtil.testRegexLatin1({
111             chaine : chaine ,
112             minLength : 1,
113             maxLength : 255
114         }) == true)
115         {
116             return true ;
117         } else {
118             return "Le pays , obligatoire , ne contient que des lettres , "
119                 + "tirets et guillemets ." ;
120         }
121     },
122     labelText : "Pays"
123 }
124 } // fin de l'objet attributesPatterns
125 ); // fin de l'invocation de myApp.metier.createModuleMetier

```

### 2.6.3 Fabrique G n rique d'Instances M tier (Version 1)

Nous cr ons ensuite, via un *pattern* Module, une fabrique *g n rique* (ou encore *fabrique abstraite*) d'instances d'objets m tier. Cette fabrique est g n rique en ce sens qu'elle peut servir de fabrique pour n'importe quel module m tier qui impl mente les m thodes `getAttributeList()`, `getLabelText()` et `testRegex()`, comme les modules m tier g n r  en utilisant le code source 2.17, comme illustr  dans le code source 2.18. Nous verrons par exemple, dans la partie 2.6.4, comment utiliser cette m thode pour cr er tr s simplement une *fabrique concr te* d'adresses en s'appuyant sur le module `myApp.metier.adresse` (voir la partie 2.6.2).

La fabrique prend comme param tre un objet contenant des valeurs pour initialiser les attributs, effectue les tests d'expressions r guli res, et cr e deux objets priv s :

- L'objet priv  `dataAttributes` contient comme propri t s les attributs de l'instance d'objets m tier ;
- L'objet priv  `dataError` contient comme propri t s les  ventuels messages d'erreur suite aux tests (typiquement d'expression r guli re) sur la validit  des valeurs des attributs de l'instance d'objets m tier ;

Des m thodes publiques, dans l'interface du module, permettent d'acc der  , ou de modifier les donn es de l'instance. Notons que les m thodes de l'interface publique du module cr  

peuvent utiliser les méthodes du module parent en utilisant le *pattern that* (voir la partie 2.1.3).

L'interface publique des instances expose aussi les méthodes statiques (méthodes de classes), dont le code ne dépend pas des instances, mais est factorisé au niveau du module métier (ici donné par l'objet *that* suivant le *pattern that*).

Nous avons aussi ajouté la possibilité, en passant un argument *inputObj* égal à *null*, de réer une instance par défaut (*id* aléatoire et autres attributs vides) Ceci permet par exemple d'initialiser un formulaire vide pour créer une nouvelle instance.

Code Source 2.19 : /pattern-fonct/ex07-fabriqueObjetMetier.js

```

1  /** @description Fabrique qui crée des objets d'un module métier, suivant le "
    pattern module".
2  * Le paramètre spec de notre fonction est un objet contenant les valeurs des
    attributs
3  * de l'instance à créer.
4  *
5  * Pour être utilisée, cette fabrique doit être ajoutée en tant que méthode
6  * à un module métier comportant déjà dans son interface publique les méthodes
7  * getAttributeList, getLabelText et testRegex comme les modules
8  * générés par la méthode myApp.Metier.createModuleMetier(.,.).
9  *
10 * @method createInstanceGenerique
11 * @augments myApp.metier
12 * @param {Object} inputObj - spécification des valeurs des attributs d'une
    instance de module métier
13 * @param {string} inputObj.id - identifiant unique de l'instance
14 * @param {string/number} inputObj.attributeName - valeur à affecter à l'
    attribut attributeName
15 * (exemple pour une adresse : inputObj.numeroRue, inputObj.
    codePostal, etc.
16 */
17 myApp.addModule.apply(myApp.metier, [ "createInstanceGenerique", function(
    inputObj){
18
19     //////////////////////////////////////
20     // Propriétés et méthodes "statiques" privées
21
22     /** @description contient les valeurs des attributs de l'instance,
        initialement vide
23     * @member
24     * @private
25     */
26     var dataAttributes = {};
27     /** @description contient les messages d'erreur associés aux attributs de l'
        instances
28     * @member
29     * @private
30     */
31     var dataError = {};
32
33     // Application du pattern "that"
34     // Si une fonction locale (comme setAttributeAndError plus loin)
35     // veut utiliser une méthode du module contenant concrètement notre fabrique,
36     // (par exemple myApp.metier.adresse.testRegex)
37     // Il doit appeler (par exemple) this.setAttributeAndError.
38     // Mais si la méthode setAttributeAndError est appelée dans

```

```

39 // l'interface publique des instances (publicInterfaceInstance) ci-dessous,
40 // "this" référerà à cet autre objet (ici publicInterfaceInstance).
41 // Une solution consiste à mémoriser l'objet this dans une variable that,
42 // et d'appeler la méthode that.testRegex et non pas this.testRegex
43 // dans la fonction locale setAttributeAndError ci-dessous.
44 var that = this; // exemple : this = myApp.metier.adresse
45
46 /** @description Ajoute un message d'erreur associé à un attribut
47  * @method addError
48  * @private
49  */
50 var addError = function(attributeName, message){
51     // Ajout d'une propriété
52     dataError[attributeName] = message;
53 }
54
55 /** @description Setter : initialise la valeur pour un attribut d'une instance.
56  * En cas d'erreur un message pour cet attribut est est ajouté dans dataError.
57  * En l'absence d'erreur, une éventuelle erreur précédente est effacée.
58  * @method inputObj.
59  * @private
60  */
61 var setAttributeAndError = function(attributeName, value){
62     var resultTestRegex = that.testRegex(attributeName, value);
63     // On initialise l'attribut de l'instance
64     dataAttributes[attributeName] = value;
65     // Si la validation par expression régulière est passée
66     if (resultTestRegex === true){
67         // On efface une vieille erreur éventuelle
68         delete dataError[attributeName];
69     }else{
70         // On initialise l'attribut de l'objet des erreurs.
71         // avec le message d'erreur.
72         addError(attributeName, "Attribut " + value +
73             " invalide : " + resultTestRegex);
74     }
75 }
76
77 // Initialisation des attributs de l'instance.
78 // Si l'objet en argument est null, on construit une instance par défaut
79 if (inputObj !== null){
80     // Parcours des propriétés de getAttributeList()
81     // qui correspondent aux attributs de l'instance à créer
82     for (var i = 0 ; i < this.getAttributeList().length ; ++i){
83         var attributeName = this.getAttributeList()[i];
84         // Utilisation du setter pour initialiser l'attribut
85         setAttributeAndError(attributeName, inputObj[attributeName]);
86     }
87 }else{
88     // On initialise les valeurs des attributs à ""
89     for (var i = 0 ; i < this.getAttributeList().length ; ++i){
90         var attributeName = this.getAttributeList()[i];
91         dataAttributes[attributeName] = "";
92     }
93 }
94

```

```

95  //////////////////////////////////////
96  // Interface publique du module
97
98  /** @description Interface publique des instances du module métier.
99   * Fournit les méthodes pour manipuler l'instance (accesseurs, setters...)
100  */
101  var publicInterfaceInstance = {
102    /** @description Retourne le module avec les méthodes "statiques"
103     * (comme l'accès direct à la liste des propriétés ou les tests regex)
104     * @return {Object} le module myApp.metier.moduleName
105     */
106    getModule : function() {
107      return that;
108    },
109
110    /** @description Accesseur pour tous les membres privés d'instance.
111     * @method getAttribute
112     * @public
113     * @param {string} attributeName - nom de l'attribut attendue d'une
114       instance
115     * @return {string} la valeur de l'attribut ou undefined en cas de nom d'
116       attribut inconnu.
117     */
116    getAttribute : function(attributeName) {
117      return dataAttributes[attributeName];
118    },
119
120    /** @description Setter :
121     * initialise la valeur pour un attribut d'une instance après un test.
122     * En cas d'erreur, un message pour cet attribut est ajouté dans dataError.
123     * En l'absence d'erreur, une éventuelle erreur précédente est effacée.
124     * Il s'agit d'un simple alias vers la fonction privée setAttributeAndError.
125     * @method setAttribute
126     * @public
127     * @param {string} attributeName - nom de l'attribut attendue d'une instance
128     * @param {string} value - valeur à prendre pour l'attribut attendu d'une
129       instance
130     * @return {boolean} true s'il y a au moins une erreur, false sinon
131     */
131    setAttribute : setAttributeAndError,
132
133    /** @description permet de savoir si un des attributs au moins comporte une
134      erreur.
135     * @return {boolean} true s'il y a au moins une erreur, false sinon
136     */
136    hasError : function() {
137      for (var attributeName in dataError) {
138        if (dataError.hasOwnProperty(attributeName)) {
139          return true;
140        }
141      }
142      return false;
143    },
144
145    /** @description Donne l'accès au message d'erreur d'un attribut (s'il existe
146      ).

```

```

146  * @param {string} attributeName – nom d'attribut d'une instance de module mé
    tier
147  * @return {string|undefined} le message d'erreur pour un attribut s'il
    existe
148  *
    ou undefined en l'absence d'erreur
149  */
150  getErrorMessage : function(attributeName){
151      return dataError[attributeName];
152  },
153
154  /** @description Récupère la liste des noms d'attributs qui ont une erreur
155  * @return {string[]} tableau des noms d'attributs qui comportent une erreur
    .
156  */
157  getErrorList : function(){
158      var errorList = [];
159      for (var attributeName in dataError){
160          if (dataError.hasOwnProperty(attributeName)){
161              errorList.push(attributeName);
162          }
163      }
164      return errorList;
165  },
166
167  //////////////////////////////////////
168  // Ajout des méthodes statiques du module dans l'interface des instances
169  //////////////////////////////////////
170
171  /** @description Renvoie la liste des noms d'attributs des instances.
172  * @method getAttributeList
173  */
174  getAttributeList : function(){
175      return that.getAttributeList();
176  },
177
178  /** @description Renvoie le texte de description d'un attribut des instances
    .
179  * @method getLabelText
180  * @param {string} attributeName – nom d'un attribut
181  * @return {string} le texte de description courte du champ
182  */
183  getLabelText : function(attributeName){
184      return that.getLabelText(attributeName);
185  },
186
187  /** @description Expose le test d'expression régulière des attributs
188  * des instances.
189  * Peut être utilisée pour le filtrage des données d'un formulaire.
190  * @method testRegex
191  * @param {string} attributeName – nom d'un attribut
192  * @param {string} value – valeur pour initialiser l'attribut
193  * @return {boolean|string} true si la chaîne est un code postal valide ,
194  * un message d'erreur sinon.
195  */
196  testRegex : function(attributeName, value){
197      return that.testRegex(attributeName, value);

```

```

198     },
199
200     /** @description Fabrique qui crée des objets d'un module métier, suivant le "
        pattern module".
201     * Le paramètre spec de notre fonction est un objet contenant les valeurs des
        attributs
202     * de l'instance à créer.
203     * @param {Object} inputObjet - spécification des valeurs des attributs d'une
        instance de module métier
204     * @param {string} inputObjet.id - identifiant unique de l'instance
205     * @param {string} inputObjet.attributeName - valeur à affecter à l'attribut
        attributeName
206     * (exemple pour une adresse : inputObj.numeroRue, inputObj
        .codePostal, etc.
207     */
208     createInstance : function(inputObjet){
209         return that.createInstance(inputObjet);
210     },
211
212     /** @description Crée une copie de l'objet à l'identique.
213     * @returns {Object} une instance avec des valeurs d'attributs identique à
        this.
214     */
215     clone : function(){
216         return this.createInstance(dataAttributes);
217     }
218 }; // fin de publicInterfaceInstance
219
220 return publicInterfaceInstance;
221
222 } // fin de la méthode createInstance
223 ]; // fin de l'appel "apply" de la méthode myApp.addModule

```

### 2.6.4 Exemple : La Fabrique du Module adresse

La fabrique d'instances d'adresse, comme toutes les autres fabriques concrètes pour les instances d'objets métier, est simplement une copie de la fabrique générique d'objets métier vue dans la partie 2.6.3, qui est simplement placée dans le sous-module `myApp.metier.adresse`.

Ainsi, lors de l'invocation de cette fabrique, les méthodes `getAttributeList()`, `getLabelText()` et `that.testRegex()` font référence aux méthodes éponymes du module `myApp.metier.adresse`, ce qui permet à la fabrique, dont le code est générique, de travailler concrètement sur des adresses.

Code Source 2.20 : /pattern-fonct/ex08-fabriqueAdresse.js

```

1  /**
2   * Fabrique qui crée des objets représentant des adresse, suivant le "pattern
        module".
3   * Le paramètre spec de notre fonction est un objet contenant les propriétés d'
        une adresse à créer.
4   *
5   * @method createInstance
6   * @augments myApp.metier.adresse
7   * @param {Object} inputObj - spécification des propriétés d'une instance d'
        adresse

```

```

8  * @param {string} inputObj.id - identifiant unique de l'instance
9  * @param {string} inputObj.numeroRue - numero de rue
10 * @param {string} inputObj.rue - nom de rue
11 * @param {string} inputObj.complementAddr - complément d'adresse (lieu dut, bâ
    timent, résidence, etc.)
12 * @param {string} inputObj.codePostal - code postal
13 * @param {string} inputObj.ville - nom de ville
14 * @param {string} inputObj.pays - nom de pays
15 */
16 myApp.addModule.apply(myApp.metier.adresse, ["createInstance",
17                                           myApp.metier.createInstanceGenerique]); //
    fin de l'appel "apply" de la mé
    thode myApp.addModule

```

Le fichier *HTML* réalise le test de création d'une instance et d'utilisation de *setters*, et affiche les données et les erreurs obtenues.

| Données :   | Erreurs :   |
|---|---|
| <b>Identifiant</b> : 04abf85bc9<br><b>Numéro</b> : 2 bis<br><b>rue/place</b> : Rue de l'a Paix<br><b>Lieu dit, Bâtiment, BP</b> : undefined<br><b>Code Postal</b> : 630000<br><b>Ville</b> : Clermont-Ferrand<br><b>Pays</b> : France 2                     | <b>Identifiant</b> : undefined<br><b>Numéro</b> : undefined<br><b>rue/place</b> : undefined<br><b>Lieu dit, Bâtiment, BP</b> : Propriété undefined invalide : le complément d'adresse ne contient que des lettres, tirets et guillemets ou chiffres.<br><b>Code Postal</b> : Propriété 630000 invalide : Le code postal doit comporter 5 chiffres décimaux.<br><b>Ville</b> : undefined<br><b>Pays</b> : Propriété France 2 invalide : Le pays, obligatoire, ne contient que des lettres, tirets et guillemets. |
| <b>Données :</b><br><b>Identifiant</b> : 04abf85bc9<br><b>Numéro</b> : @#*m<br><b>rue/place</b> : Rue de l'a Paix<br><b>Lieu dit, Bâtiment, BP</b> : "Bâtiment 3D"<br><b>Code Postal</b> : 63000<br><b>Ville</b> : Clermont-Ferrand<br><b>Pays</b> : France | <b>Erreurs :</b><br><b>Identifiant</b> : undefined<br><b>Numéro</b> : Propriété @#*m invalide : Le numéro de la rue contient au plus 15 caractères, lettres, tirets et guillemets ou chiffres.<br><b>rue/place</b> : undefined<br><b>Lieu dit, Bâtiment, BP</b> : undefined<br><b>Code Postal</b> : undefined<br><b>Ville</b> : undefined<br><b>Pays</b> : undefined  |

FIGURE 2.6 : Illustration du code source 2.21

Code Source 2.21 : `/pattern-fonct/ex06-moduleMetierAdresseTest.js` (cf. Fig 2.6)

```

1  /** Module de test avec une méthode d'affichage d'une instance
2   * d'adresse avec ses éventuelles erreurs.
3   * @module myApp.test
4   * @method myApp.test.testAfficheAdresse
5   * @public
6   * @param {Object} adresse - instance d'objet métier du module "adresse"
7   */
8  myApp.addModule("test", {
9    testAfficheAdresse : function(adresse){
10      // Code HTML pour l'affichage de l'instance :
11      var codeHTML = "<div><span><h2>Données </h2>" +
12                    "<strong>" + adresse.getLabelText('id') +
13                    "</strong>" + adresse.getAttribute('id') + "<br/>" +
14                    "<strong>" + adresse.getLabelText('numeroRue') +
15                    "</strong>" + adresse.getAttribute('numeroRue') + "<br/>" +
16                    "<strong>" + adresse.getLabelText('rue') +

```

```

17     " </strong> " + adresse.getAttribute( 'rue' ) + "<br />" +
18     "<strong>" + adresse.getLabelText( 'complementAddr' ) +
19     " </strong> " + adresse.getAttribute( 'complementAddr' ) + "<br />" +
20     "<strong>" + adresse.getLabelText( 'codePostal' ) +
21     " </strong> " + adresse.getAttribute( 'codePostal' ) + "<br />" +
22     "<strong>" + adresse.getLabelText( 'ville' ) +
23     " </strong> " + adresse.getAttribute( 'ville' ) + "<br />" +
24     "<strong>" + adresse.getLabelText( 'pays' ) +
25     " </strong> " + adresse.getAttribute( 'pays' ) +
26     "</span>";
27
28     // variante en énumérant automatiquement les propriétés
29     codeHTML += "<span><h2>Erreurs </h2>" +
30     "<strong>" + adresse.getLabelText( 'id' ) +
31     " </strong>" + adresse.getErrorMessage( 'id' ) + "<br />" +
32     "<strong>" + adresse.getLabelText( 'numeroRue' ) +
33     " </strong> " + adresse.getErrorMessage( 'numeroRue' ) + "<br />" +
34     "<strong>" + adresse.getLabelText( 'rue' ) +
35     " </strong> " + adresse.getErrorMessage( 'rue' ) + "<br />" +
36     "<strong>" + adresse.getLabelText( 'complementAddr' ) +
37     " </strong> " + adresse.getErrorMessage( 'complementAddr' ) + "<br />" +
38     "<strong>" + adresse.getLabelText( 'codePostal' ) +
39     " </strong> " + adresse.getErrorMessage( 'codePostal' ) + "<br />" +
40     "<strong>" + adresse.getLabelText( 'ville' ) +
41     " </strong> " + adresse.getErrorMessage( 'ville' ) + "<br />" +
42     "<strong>" + adresse.getLabelText( 'pays' ) +
43     " </strong> " + adresse.getErrorMessage( 'pays' ) +
44     "</span></div>";
45     return codeHTML;
46 } // fin de la méthode testAfficheAdresse
47 } // fin du module myApp.test
48 ); // fin de l'invocation de myApp.addModule
49
50 /** @description Programme principal qui construit les données
51  * et génère la vue.
52  * @method myApp.mainFunction
53  * @public
54  */
55 myApp.addModule( "mainFunction", function() {
56     // création d'une instance
57     var adresse = myApp.metier.adresse.createInstance({
58         id : "04abf85bc9",
59         numeroRue : "2 bis",
60         rue : "Rue de l'a Paix",
61         // oubli du champs complementAddr
62         codePostal : "63000",
63         ville : "Clermont-Ferrand",
64         pays : "France 2"
65     });
66
67     var codeHTML = this.test.testAfficheAdresse( adresse );
68
69     // Test de setter
70     adresse.setAttribute( "complementAddr", "\"Bâtiment 3D\"" );
71     adresse.setAttribute( "codePostal", "63000" );
72     adresse.setAttribute( "pays", "France" );

```



```

73  adresse.setAttribute("numeroRue", "@#*m");
74
75  codeHTML += this.test.testAfficheAdresse(adresse);
76
77  // Test de clonage :
78  codeHTML += this.test.testAfficheAdresse(adresse.clone());
79
80  // Utilisation de la valeur retournée pour générer la vue
81  document.getElementById("paragrapheResultat").innerHTML = codeHTML;
82  });
83
84  // Exécution de la méthode mainFunction
85  myApp.mainFunction();

```

Code Source 2.22 : Fichiers *JS* inclus dans ex06-moduleMetierAdresseTest.html

```

1  <!-- Création de l'application vide avec deux méthodes -->
2  <script src="ex04-structureApplication.js"></script>
3  <!-- Création de sous-module regexUtil de myApp.metier -->
4  <script src="/ex05-modulePatternRegex.js"></script>
5  <!-- Création de sous-module adresse de myApp.metier -->
6  <script src="/ex06-createModuleMetier.js"></script>
7  <!-- Création d'une méthode fabrique générique d'objets métier -->
8  <script src="/ex07-fabriqueObjetMetier.js"></script>
9  <!-- Création de sous-module adresse de myApp.metier -->
10 <script src="/ex06-moduleMetierAdresse.js"></script>
11 <!-- Création d'une méthode fabrique d'adresse de myApp.metier.adresse -->
12 <script src="/ex08-fabriqueAdresse.js"></script>
13 ---
14 <!-- Ajout d'une fonction de test, d'une méthode "main", et exécution -->
15 <script src="/ex06-moduleMetierAdresseTest.js"></script>

```

## 2.6.5 Utilisation : Création d'un Module `myApp.view.adresse`

Nous ajoutons, dans un module `myApp.view.adresse`, des méthodes pour générer le code *HTML* d'une adresse, au format compact (sur une ligne) ou au format développé (avec le détail des labels des attribut).

Code Source 2.23 : /pattern-fonct/ex09-adresseView.js

```

1  // Création d'un module myApp.view et d'un sous-module myApp.view.adresse
2  myApp.addModule("view", {adresse: {}});
3
4  /**
5   * Méthode de génération de code HTML pour une instance d'adresse.
6   * Pour chaque propriété attendue d'une adresse, la description de la propriété
7   *   et sa valeur sont affichées.
8   *
9   * @method getHtmlDevelopped
10  * @augments myApp.view.adresse
11  * @param {Object} adresse - spécification des propriétés d'une instance d'
12  *   adresse
13  * @param {string} adresse.id - identifiant unique de l'instance
14  * @param {string} adresse.numeroRue - numero de rue
15  * @param {string} adresse.rue - nom de rue

```

```

14  * @param {string} adresse.complementAddr - complément d'adresse (lieu dut, bâ
    timent, résidence, etc.)
15  * @param {string} adresse.codePostal - code postal
16  * @param {string} adresse.ville - nom de ville
17  * @param {string} adresse.pays - nom de pays
18  */
19  myApp.addModule.apply(myApp.view.adresse, [ "getHtmlDevelopped", function(adresse
    ){
20      var htmlCode = "";
21
22      var moduleAdresse = adresse.getModule();
23
24      if (adresse.getAttribute( 'numeroRue ')){
25          htmlCode += "<span class=\"adresseItem\">" + moduleAdresse.getLabelText( '
                numeroRue ' ) + "Énbsp; </span> " +
26          adresse.getAttribute( 'numeroRue ' ) + "<br />";
27      }
28
29      htmlCode += "<span class=\"adresseItem\">" + moduleAdresse.getLabelText( 'rue ' )
                + "Énbsp; </span> " +
30          adresse.getAttribute( 'rue ' ) + "<br />";
31
32      if (typeof adresse.getAttribute( 'complementAddr ' ) === "string" &&
33          adresse.getAttribute( 'complementAddr ' ) !== ""){
34          htmlCode += "<span class=\"adresseItem\">" + moduleAdresse.getLabelText( '
                complementAddr ' ) + "Énbsp; </span> " +
35          adresse.getAttribute( 'complementAddr ' ) + "<br />";
36      }
37
38      htmlCode += "<span class=\"adresseItem\">" + moduleAdresse.getLabelText( '
                codePostal ' ) + "Énbsp; </span> " +
39          adresse.getAttribute( 'codePostal ' ) + "<br /> " +
40          "<span class=\"adresseItem\">" + moduleAdresse.getLabelText( 'ville ' ) + "É
                nbsp; </span> " +
41          adresse.getAttribute( 'ville ' ) + "<br /> " +
42          "<span class=\"adresseItem\">" + moduleAdresse.getLabelText( 'pays ' ) + "É
                nbsp; </span> " +
43          adresse.getAttribute( 'pays ' ) + "<br />";
44
45      if (adresse.hasError()){
46          var errorList = adresse.getErrorList();
47          htmlCode += "<strong>Certains champs ont une erreur </strong><br />";
48          for (var i=0 ; i<errorList.length ; i++){
49              if (i > 0){
50                  htmlCode += ", ";
51              }
52              htmlCode += errorList[i];
53          }
54      }
55
56      return htmlCode;
57  }]);
58
59  /**
60  * Méthode de génération de code HTML pour une instance d'adresse.
61  * L'adresse est affichée sur une ligne, sans mention des erreurs.

```

```

62 *
63 * @method getHtmlDevelopped
64 * @augments myApp.view.adresse
65 * @param {Object} adresse — spécification des propriétés d'une instance d'
    adresse
66 * @param {string} adresse.id — identifiant unique de l'instance
67 * @param {string} adresse.numeroRue — numero de rue
68 * @param {string} adresse.rue — nom de rue
69 * @param {string} adresse.complementAddr — complément d'adresse (lieu dut, bâ
    timent, résidence, etc.)
70 * @param {string} adresse.codePostal — code postal
71 * @param {string} adresse.ville — nom de ville
72 * @param {string} adresse.pays — nom de pays
73 */
74 myApp.addModule.apply(myApp.view.adresse, [ "getHtmlCompact", function(adresse){
75     var htmlCode = "";
76
77     if (adresse.getAttribute( 'numeroRue ')){
78         htmlCode += adresse.getAttribute( 'numeroRue ') + ", ";
79     }
80
81     htmlCode += adresse.getAttribute( 'rue ') + ", ";
82     if (adresse.getAttribute( 'complementAddr ')){
83         htmlCode += adresse.getAttribute( 'complementAddr ') + ", ";
84     }
85     htmlCode += adresse.getAttribute( 'codePostal ') + " " +
86         adresse.getAttribute( 'ville ') + ", " +
87         adresse.getAttribute( 'pays ');
88     return htmlCode;
89 }]);

```

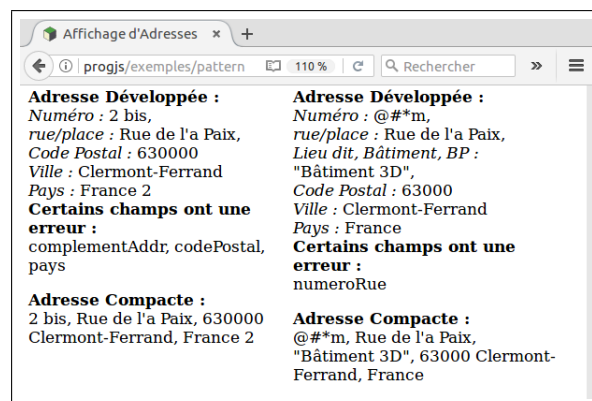


FIGURE 2.7 : Illustration du code source 2.24

Code Source 2.24 : /pattern-fonct/ex09-adresseViewTest.js (cf. Fig 2.7)

```

1 /** Module de test avec une méthode d'affichage d'une instance
2  * d'adresse avec les méthodes du package myApp.view.adresse
3  * @module myApp.test
4  * @method myApp.test.testAfficheAdresse
5  * @public
6  * @param {Object} adresse — instance d'objet métier du module "adresse"
7  */

```

```

8 myApp.addModule( "test", {
9     testAfficheAdresse : function(adresse){
10         return "<span style=\`width :260px; display : inline-block; vertical-align :
            top ;\`>" +
11             "<p<strong>Adresse Développée </strong><br />" +
12             myApp.view.adresse.getHtmlDevelopped(adresse) + "</p>" +
13             "<p<strong>Adresse Compacte </strong><br />" +
14             myApp.view.adresse.getHtmlCompact(adresse) + "</p>" +
15             "</span>";
16     } // fin de la méthode testAfficheAdresse
17 } // fin du module myApp.test
18 ); // fin de l'invocation de myApp.addModule
19
20
21 /** @description Programme principal qui construit les données
22  * et génère la vue.
23  * @method myApp.mainFunction
24  * @public
25  */
26 myApp.addModule( "mainFunction", function() {
27     // création d'une instance
28     var adresse = myApp.metier.adresse.createInstance({
29         id : "04abf85bc9",
30         numeroRue : "2 bis",
31         rue : "Rue de l'a Paix",
32         // oubli du champs complementAddr
33         codePostal : "63000",
34         ville : "Clermont-Ferrand",
35         pays : "France 2"
36     });
37
38     var codeHTML = this.test.testAfficheAdresse(adresse);
39
40     adresse.setAttribute( "complementAddr", "\`Bâtiment 3D\`" );
41     adresse.setAttribute( "codePostal", "63000" );
42     adresse.setAttribute( "pays", "France" );
43     adresse.setAttribute( "numeroRue", "@#*n" );
44
45     codeHTML += this.test.testAfficheAdresse(adresse);
46     // Utilisation de la valeur retournée pour générer la vue
47     document.getElementById( "paragrapheResultat" ).innerHTML = codeHTML;
48 };
49
50 // Exécution de la méthode mainFunction
51 myApp.mainFunction();

```

Code Source 2.25 : Fichiers *JS* inclus dans ex09-adresseViewTest.html

```

1 <!-- Création de l'application vide avec deux méthodes -->
2 <script src="ex04-structureApplication.js"></script>
3 <!-- Création de sous-module regexUtil de myApp.metier -->
4 <script src="./ex05-modulePatternRegex.js"></script>
5 <!-- Création de sous-module adresse de myApp.metier -->
6 <script src="./ex06-createModuleMetier.js"></script>
7 <!-- Création d'une méthode fabrique générique d'objets métier -->
8 <script src="./ex07-fabriqueObjetMetier.js"></script>

```

```

9  <!-- Création de sous-module adresse de myApp.metier -->
10 <script src="/ex06-moduleMetierAdresse.js"></script>
11 <!-- Création d'une méthode fabrique d'adresse de myApp.metier.adresse -->
12 <script src="/ex08-fabriqueAdresse.js"></script>
13 <!-- Création de fonctions d'affichage dans myApp.metier.view.adresse -->
14 <script src="/ex09-adresseView.js"></script>
15 --
16 <!-- Ajout d'une fonction de test, d'une méthode "main", et exécution -->
17 <script src="/ex09-adresseViewTest.js"></script>

```

## 2.7 Interface Générique de Objets métier

Il est possible de tester l'affichage des objets métier de manière générique, sans supposer qu'il s'agit particulièrement d'une adresse, sur la base de l'implémentation d'une interface. Nous devons d'abord créer un outil pour vérifier l'implémentation d'une interface par des objets *JavaScript*.

### 2.7.1 Implémentation d'interfaces en *JavaScript*

Voici une classe `Interface`, qui possède comme attribut un `Array` de noms de méthodes, et qui permet de vérifier qu'un objet possède bien des méthodes avec les noms correspondants. Notons que nous ne vérifions pas que les méthodes correspondent bien à un prototype déterminé, mais seulement que les noms de méthodes sont présents.

Code Source 2.26 : `/pattern-fonct/ex11-interfaceImplementation.js`

```

1  /**
2   * Définit une "interface", avec un nom et un ensemble de méthodes.
3   * Ceci nous permettra de vérifier qu'un certain nombre d'opérations
4   * sont présentes dans un objet JavaScript.
5   *
6   * @constructor Interface
7   * @param {string} name - nom de l'interface
8   * @param {string[]} methods - tableau contenant les noms des méthodes de l'
   *   interface.
9   */
10 var Interface = function(methods) {
11
12   if (methods.length === undefined) {
13     throw {
14       name: "IllegalArgument",
15       message: "Une interface nécessite un array (ou array-like) de noms de méthodes."
16     };
17   }
18
19   // Création d'une propriété pour stocker le nom de l'interfac
20   // Création d'un tableau pour stocker les noms de méthodes
21   this.methods = [];
22   // pour chaque nom de méthode
23   for (var i = 0 ; i < methods.length ; ++i) {
24     // Vérification de type
25     if (typeof methods[i] !== 'string') {

```

```

26     throw {
27         name : "IllegalArgument",
28         message : "Les noms de méthodes d'une interface doivent être de type
                string."
29     };
30 }
31 // Ajout du nom de méthode
32 this.methods.push(methods[i]);
33 }
34 };
35
36 /**
37  * Vérifie qu'un objet "implémente une interface", en ce sens qu'il comporte
38  * un certain nombre de méthodes (propriétés de type fonction) qui ont les
39  * mêmes noms que les méthodes de l'interface.
40  *
41  * @method isImplementedBy
42  * @param {Object} objet — objet qui doit implémenter l'interface.
43  * @return {boolean/string} true si l'objet comporte toutes les méthodes de l'
44  *     interface,
45  *     un message d'erreur indiquant une méthode qui n'est pas présente dans l'objet
46  *     sinon.
47  */
48 Interface.prototype.isImplementedBy = function(objet) {
49     // Pour chaque nom de méthode
50     for (var i = 0 ; i < this.methods.length ; ++i) {
51         var methodName = this.methods[i];
52         // Si l'objet n'a pas de propriété de ce nom qui soit de type fonction
53         if (!objet[methodName] || typeof objet[methodName] !== 'function') {
54             return "L'objet n'implémente pas la méthode " + methodName;
55         }
56     }
57     return true;
58 };

```

Voici un exemple dans lequel nous définissons une interface attendue de nos modules métier. Cette interface contient deux types de méthodes :

1. L'interface attendue des instances du module métier permet de tester la présence d'un certain nombre de méthodes communes à toutes les instances (similaires à des méthodes de classe), qui sont définies dans le prototype des instances ;
2. L'interface attendue des instances permet de s'assurer de la présence d'un certain nombre de méthodes sur les instances, qui sont définies soit au niveau du prototype (si le code source est indépendant de l'instance) ou au niveau de l'instance (si besoin de personnaliser le code).

## 2.7.2 Interface des instances de modules métier

Code Source 2.27 : `/pattern-fonct/ex11-interfaceImplementationMetier.js`

```

1  /** @description Test si un objet implémente les interfaces génériques
2   * pour les objets métiers suivant notre modlisation.
3   *

```

```

4  * @method testInterfaceImplementation
5  * @augments myApp.metier
6  * @public
7  * @param {Object} instance – Instance d'objet métier supposée
8  * @param {function} instance.getModule – retourne le module métier (méthodes
   *    statiques)
9  * @param {function} instance.getAttribute
10 * @param {function} instance.setAttribute
11 * @param {function} instance.hasError
12 * @param {function} instance.getErrorMessage
13 * @param {function} instance.getErrorList
14 *
15 * @param {function} instance.getModule().getAttributeList
16 * @param {function} instance.getModule().getLabelText
17 * @param {function} instance.getModule().testRegex
18 * @param {function} instance.getModule().createInstance
19 *
20 * @return
21 */
22 myApp.addModule.apply(myApp.metier, [ "testInterfaceImplementation", function(
   instance){
23
24 // Définition de l'interface commune aux modules métier (adresse, personne,
   etc.)
25 var metierCommonMethods = new Interface([
26 // 1) Méthodes statiques (résultats indépendant de l'instance)
27 "getAttributeList", "getLabelText", "testRegex", "createInstance",
28 // 2) Méthodes d'instance (résultats dépendant de l'instance)
29 "getModule", "getAttribute", "setAttribute", "hasError", "getErrorMessage",
   "getErrorList"
30 ]);
31
32 // création d'une instance (en l'occurrence une adresse)
33 var monObjet = myApp.metier.adresse.createInstance({
34   id: "04abf85bc9",
35   numeroRue: "2 bis",
36   rue: "Rue de l'a Paix",
37   complementAdresse: "Bâtiment 3D",
38   codePostal: "63000",
39   ville: "Clermont-Ferrand",
40   pays: "France"
41 });
42
43 return metierCommonMethods.isImplementedBy(monObjet);
44 }]);

```

Code Source 2.28 : /pattern-fonct/ex11-interfaceImplementationTest.js

```

1  /** @description Programme principal qui construit les données
2  * et génère la vue.
3  * @method myApp.mainFunction
4  * @public
5  */
6  myApp.addModule( "mainFunction", function(){
7    // création d'une instance
8    var monObjet = myApp.metier.adresse.createInstance({

```

```

9      id : "04abf85bc9",
10     numeroRue : "2 bis",
11     rue : "Rue de l'a Paix",
12     complementAdresse : "Bâtiment 3D",
13     codePostal : "63000",
14     ville : "Clermont-Ferrand",
15     pays : "France"
16   });
17
18   var testInterfacesMetier = myApp.metier.testInterfaceImplementation(monObjet);
19   var codeHTML;
20   if (testInterfacesMetier === true){
21     codeHTML = "<p>L'objet semble bien implémenter les méthodes requises.</p>";
22   }else{
23     codeHTML = "<p>" + testInterfacesMetier + "</p>"
24   }
25   // Utilisation de la valeur retournée pour générer la vue
26   document.getElementById("paragrapheResultat").innerHTML = codeHTML;
27 });
28
29 // Exécution de la méthode mainFunction
30 myApp.mainFunction();

```

”

Code Source 2.29 : Fichiers *JS* inclus dans ex11-interfaceImplementationTest.html

```

1  <!-- Création de l'application vide avec deux méthodes -->
2  <script src="ex04-structureApplication.js"></script>
3  <!-- Création de sous-module regexUtil de myApp.metier -->
4  <script src="/ex05-modulePatternRegex.js"></script>
5  <!-- Création de sous-module adresse de myApp.metier -->
6  <script src="/ex06-createModuleMetier.js"></script>
7  <!-- Création d'une méthode fabrique générique d'objets métier -->
8  <script src="/ex07-fabriqueObjetMetier.js"></script>
9  <!-- Création de sous-module adresse de myApp.metier -->
10 <script src="/ex06-moduleMetierAdresse.js"></script>
11 <!-- Création d'une méthode fabrique d'adresse de myApp.metier.adresse -->
12 <script src="/ex08-fabriqueAdresse.js"></script>
13 <!-- Création de fonctions d'affichage dans myApp.metier.view.adresse -->
14 <script src="/ex09-adresseView.js"></script>
15 <!-- Classe de vérification de l'implémentation d'interfaces -->
16 <script src="/ex11-interfaceImplementation.js"></script>
17 <!-- Classe de vérification de l'implémentation d'interfaces -->
18 <script src="/ex11-interfaceImplementationMetier.js"></script>
19 --
20 <!-- Ajout d'une fonction de test, d'une méthode "main", et exécution -->
21 <script src="/ex11-interfaceImplementationTest.js"></script>

```

### 2.7.3 Exemple d'utilisation : Méthode d'affichage générique

Code Source 2.30 : /pattern-fonct/ex12-objetMetierView.js

```

1 // Création d'un module myApp.view et d'un sous-module myApp.view.adresse
2 myApp.addModule("view", { adresse : {} });
3

```



```

4  /**
5   * Méthode de génération de code HTML pour une instance d'objet métier.
6   * Pour chaque attribut d'une instance, la description de la propriété
7   * et sa valeur sont affichées.
8   *
9   * @method getHtmlDevelopped
10  * @augments myApp.view
11  * @public
12  * @param {Object} instance - instance d'objet métier qui implémente :
13  *                             "getModule", "getAttribute", "setAttribute",
14  *                             "hasError", "getErrorMessage", "getErrorList"
15  *                             ainsi que les méthodes statiques :
16  *                             "getAttributeList", "getLabelText", "testRegex",
17  *                             "createInstance", "getModule".
18  */
19  myApp.addModule.apply(myApp.view, ["getHtmlGeneric", function(instance){
20    var testInterfacesMetier = myApp.metier.testInterfaceImplementation(instance);
21    if (testInterfacesMetier !== true){
22      // Retourner un message d'erreur au lieu du code HTML de l'instance
23      return testInterfacesMetier;
24    }
25    // Code HTML pour l'affichage de l'instance :
26    var codeHTML = "<div><span><h2>Données de l'instance </h2>";
27    for (var i=0 ; i < instance.getModule().getAttributeList().length ; ++i){
28      var attributeName = instance.getModule().getAttributeList()[i];
29      codeHTML += "<strong> "
30                + instance.getModule().getLabelText(attributeName) + " : "
31                + "</strong> "
32                + instance.getAttribute(attributeName) + "<br />";
33    }
34    codeHTML += "</span>";
35
36    // variante en énumérant automatiquement les propriétés
37    codeHTML += "<span><h2>Erreurs </h2>";
38    for (var i=0 ; i < instance.getModule().getAttributeList().length ; ++i){
39      var attributeName = instance.getModule().getAttributeList()[i];
40      codeHTML += "<strong> "
41                + instance.getModule().getLabelText(attributeName) + " : "
42                + "</strong> " +
43                instance.getErrorMessage(attributeName) + "<br />";
44    }
45    codeHTML += "</span></div>";
46    return codeHTML;
47  }]);

```

Code Source 2.31 : /pattern-fonct/ex12-objetMetierViewTest.js

```

1  /** Module de test avec une méthode d'affichage d'une instance
2   * d'adresse avec les méthodes du package myApp.view.adresse
3   * @module myApp.test
4   * @method myApp.test.testAfficheAdresse
5   * @public
6   * @param {Object} adresse - instance d'objet métier du module "adresse"
7   */
8  myApp.addModule("test", {
9    testAfficheInstance : function(instance){

```

```

10     return myApp.view.getHtmlGeneric(instance);
11 } // fin de la méthode testAfficheInstance
12 } // fin du module myApp.test
13 ); // fin de l'invocation de myApp.addModule
14
15
16 /** @description Programme principal qui construit les données
17  * et génère la vue.
18  * @method myApp.mainFunction
19  * @public
20  */
21 myApp.addModule("mainFunction", function(){
22     // création d'une instance
23     var adresse = myApp.metier.adresse.createInstance({
24         id: "04abf85bc9",
25         numeroRue: "2 bis",
26         rue: "Rue de l'a Paix",
27         complementAddr: "aa",
28         // oubli du champs complementAddr
29         codePostal: "63000",
30         ville: "Clermont-Fernand",
31         pays: "France 2"
32     });
33
34     var codeHTML = this.test.testAfficheInstance(adresse);
35
36     adresse.setAttribute("complementAddr", "\"Bâtiment 3D\"");
37     adresse.setAttribute("codePostal", "63000");
38     adresse.setAttribute("pays", "France");
39     adresse.setAttribute("numeroRue", "@#*m");
40
41     codeHTML += this.test.testAfficheInstance(adresse);
42
43     //alert(codeHTML);
44     // Utilisation de la valeur retournée pour générer la vue
45     document.getElementById("paragrapheResultat").innerHTML = codeHTML;
46 });
47
48 // Exécution de la méthode mainFunction
49 myApp.mainFunction();

```

”

Code Source 2.32 : Fichiers *JS* inclus dans ex12-objetMetierViewTest.html

```

1 <!-- Création de l'application vide avec deux méthodes -->
2 <script src="ex04-structureApplication.js"></script>
3 <!-- Création de sous-module regexUtil de myApp.metier -->
4 <script src="./ex05-modulePatternRegex.js"></script>
5 <!-- Création de sous-module adresse de myApp.metier -->
6 <script src="./ex06-createModuleMetier.js"></script>
7 <!-- Création d'une méthode fabrique générique d'objets métier -->
8 <script src="./ex07-fabriqueObjetMetier.js"></script>
9 <!-- Création de sous-module adresse de myApp.metier -->
10 <script src="./ex06-moduleMetierAdresse.js"></script>
11 <!-- Création d'une méthode fabrique d'adresse de myApp.metier.adresse -->
12 <script src="./ex08-fabriqueAdresse.js"></script>

```

```
13 <!-- Classe de vérification de l'implémentation d'interfaces -->
14 <script src="/ex11-interfaceImplementation.js"></script>
15 <!-- Classe de vérification de l'implémentation d'interfaces -->
16 <script src="/ex11-interfaceImplementationMetier.js"></script>
17 <!-- Création de fonctions d'affichage générique dans myApp.metier.view -->
18 <script src="/ex12-objetMetierView.js"></script>
19 --
20 <!-- Ajout d'une fonction de test, d'une méthode "main", et exécution -->
21 <script src="/ex12-objetMetierViewTest.js"></script>
```

# Chapitre 3

## Constructeurs, Prototype et *Patterns* Associés

### 3.1 Constructeurs

Un *classe* en *Javascript* se crée à partir d'un constructeur, qui est une fonction dont le nom est le nom de la classe à créer. À l'intérieur du constructeur, les propriétés de la classe sont créées et initialisées à l'aide de l'identificateur **this**. Le constructeur retourne un unique objet dont les propriétés correspondent à celles qui ont été initialisées à l'aide de l'identificateur **this**. En d'autres termes, le constructeur retourne une instance de la classe. Par convention, **les noms de constructeurs commencent par une majuscule**.

Code Source 3.1 : /pattern-proto/ex01-classeTelephone.js

```
1  /**
2   * Constructeur de téléphone. Notez la majuscule sur le nom.
3   * @constructor
4   * @param {string} tel1 - le numéro de téléphone.
5   * @param {string} [tel2] - un second numéro de téléphone.
6   */
7  var Telephone = function(tel1, /* argument optionnel */ tel2){
8
9      var checkPhone = function(tel){
10         // Test d'expression régulière après suppression des espaces et tabulations
11         if (typeof tel.libelle !== "string" || typeof tel.numero !== "string" ||
12             /^(\\+33)\\0\\[0-9\\]{9}\\$/ .test(tel.numero.replace(/\\s/g, '')) !== true){
13             throw {
14                 name: "IllegalArgumentException",
15                 message: "Numéro de téléphone \\" + tel.libelle + " : " + tel.numero + "
16                     \\" invalide"
17             }
18         };
19
20         checkPhone(tel1);
21         // Création d'un attribut de la classe
22         this.tel1=tel1;
23
24         if (tel2 !== undefined){
25             checkPhone(tel2);
26             // Création d'un attribut de la classe
```

```

27     this.tel2=tel2;
28 }
29
30 /**
31  * @method getHtml
32  * @return {string} le code HTML pour afficher une instance.
33  */
34 this.getHtml = function(){
35     var htmlCode = this.tel1.libelle + " : " + this.tel1.numero + "<br/>";
36     if (this.tel2 !== undefined){
37         htmlCode += this.tel2.libelle + " : " + this.tel2.numero + "<br/>";
38     }
39     return htmlCode;
40 };
41 }

```

Code Source 3.2 : /pattern-proto/ex01-classeTelephoneTest.js

```

1  try{
2      // Appel du constructeur avec le mot clé "new" :
3      var tel = new Telephone({ libelle : "Maison", numero : "+33 1 23 45 67 89"},
4                              { libelle : "Mobile", numero : "09 87 65 43 21"});
5      // Utilisation de la méthode getHtml()
6      var codeHTML = "<p>" + tel.getHtml() + "</p>";
7  }catch (err){
8      alert(err.message);
9  }
10 // Utilisation de "myApp.view.codeHTML" pour générer la vue
11 document.getElementById("paragrapheResultat").innerHTML = codeHTML;

```



Un constructeur doit systématiquement être employé avec le mot clé **new**. En effet, l'emploi d'un constructeur sans le mot clé **new**, qui ne génère, en soi, aucune exception ni *warning* conduit à un comportement imprévisible, généralement catastrophique. D'où l'importance de **respecter la convention que les noms de constructeurs commencent par une majuscule**, contrairement à toutes les autres fonctions ou variables.

## 3.2 Prototypes

### 3.2.1 Notion de prototype

Les méthodes de classes telles que vues jusqu'à présent ont l'inconvénient que ces méthodes sont des propriétés des objets, qui existent en autant d'exemplaires qu'il y a d'instance des objets, alors qu'elles sont constantes.

Pour éviter cela, on peut mettre les méthodes non pas directement dans l'objet, mais dans son *prototype*. Le prototype est lui-même une propriété de l'objet, mais qui est partagée entre tous les objets de la classe (il s'agit d'une variable de classe). Toutes les variables de classes doivent être créées au niveau du prototype.

Code Source 3.3 : /pattern-proto/ex02-prototype.js

1

```

2  /**
3   * Augmente la classe Telephone en ajoutant une méthode au prototype de
      Telephone
4   * @method getHtmlByLibelle
5   * @return {string} le code HTML pour afficher une instance.
6   */
7  Telephone.prototype.getNumero = function(libelle){
8      if (this.tel1.libelle.toLowerCase() === libelle.toLowerCase()){
9          return this.tel1.numero;
10     }
11     if (this.tel2 !== undefined &&
12         this.tel2.libelle.toLowerCase() === libelle.toLowerCase()){
13         return this.tel2.numero;
14     }
15     return "Numéro inexistant";
16 };

```

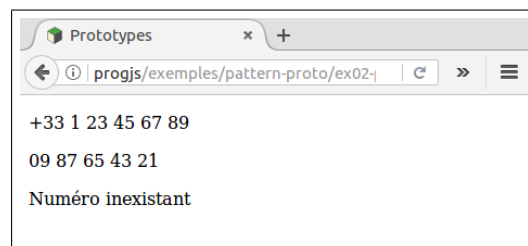


FIGURE 3.1 : Illustration du code source 3.4

Code Source 3.4 : /pattern-proto/ex02-prototypeTest.js (cf. Fig 3.1)

```

1  try{
2      // Appel du constructeur avec le mot clé "new" :
3      var tel = new Telephone({ libelle : "Maison", numero : "+33 1 23 45 67 89"},
4                              { libelle : "Mobile", numero : "09 87 65 43 21"});
5      // Utilisation de la méthode getNumero() du prototype
6      var codeHTML = "<p>" + tel.getNumero("maison") + "</p>";
7      codeHTML += "<p>" + tel.getNumero("mobile") + "</p>";
8      codeHTML += "<p>" + tel.getNumero("travail") + "</p>";
9  }catch (err){
10     alert(err.message);
11 }
12 // Utilisation de "myApp.view.codeHTML" pour générer la vue
13 document.getElementById("paragrapheResultat").innerHTML = codeHTML;

```

La méthode `Object.prototype.hasOwnProperty()` permet de tester si une propriété d'un objet existe au niveau de l'objet lui-même, ou au niveau de son prototype, ou encore du prototype de son prototype.

Le fonctionnement de la notion de prototype est le suivant : Lors d'une tentative d'accès à une propriété de l'objet, la propriété est tout d'abord recherchée au niveau des propriétés propres. Seulement si la propriété n'existe pas dans les propriétés propres, elle est ensuite recherchée dans le prototype de l'objet. Si elle n'existe pas non plus à ce niveau, la propriété est recherchée dans le prototype du prototype, et ainsi de suite...

Ce processus s'appelle la *délégation* et il permet de spécialiser les objets, en les faisant hériter des propriétés d'un prototype, tout en leur permettant de surcharger (redéfinir) les données ou

m thodes. Ceci constitue un m canisme tr s souple d'h ritage, enti rement dynamique.

### 3.2.2 Surcharge des m thodes du prototype : l'exemple de `toString`

La m thode `toString`, qui permet de convertir un objet en cha ne de caract res (par exemple pour l'afficher) a une impl mentation par d faut d finie dans le prototype de la classe `Object`. On peut la surcharger dans le prototype de notre classe `Telephone` pour changer le comportement par d faut de la m thode `toString` et mettre en forme   notre guise les num ros de t l phone.

Code Source 3.5 : `/pattern-proto/ex03-toString.js`

```
1  /**
2   * @override toString
3   * @return {string} une cha ne de caract re repr sentant l'instance de
4   *   Telephone
5   */
6  Telephone.prototype.toString = function(){
7    var texte = this.tel1.libelle + " : " + this.tel1.numero;
8    if (this.tel2 !== undefined){
9      texte += " et " + this.tel2.libelle + " : " + this.tel2.numero;
10   }
11   return texte;
12 }
```

Code Source 3.6 : `/pattern-proto/ex03-toStringTest.js`

```
1  try{
2    // Appel du constructeur avec le mot cl  "new" :
3    var tel = new Telephone({ libelle : "Maison", numero : "+33 1 23 45 67 89"},
4                          { libelle : "Mobile", numero : "09 87 65 43 21"});
5    // Utilisation implicite de la m thode toString() (conversion)
6    var codeHTML = "<p>" + tel + "</p>";
7  }catch (err){
8    alert(err.message);
9  }
10 // Utilisation de "myApp.view.codeHTML" pour g n rer la vue
11 document.getElementById("paragrapheResultat").innerHTML = codeHTML;
```

”

Code Source 3.7 : Fichiers *JS* inclus dans `ex03-toStringTest.html`

```
1  <!-- D finition du constructeur de Telephone -->
2  <script src="/ex01-classeTelephone.js"></script>
3  <!-- Red finition de la m thode toString du prototype de Telephone -->
4  <script src="/ex03-toString.js"></script>
5  --
6  <!-- Impl mentation du test et de la vue -->
7  <script src="/ex03-toStringTest.js"></script>
```

### 3.3 Modélisation de Modules Métier (version 2)

Le but de cette partie est de proposer une nouvelle modélisation pour les modules métier définis dans la partie 2.6. L'objectif est d'utiliser la notion de prototype pour éviter d'avoir autant de copies des méthodes statiques et des méthodes de classe des modules métier qu'il y a d'instance. Constatant que le code source de toutes ces méthodes est le même pour toutes les méthodes, ce code sera factorisé dans le prototype du module, qui sera lui-même accessible via le prototype des instances.

#### 3.3.1 Attributs et méthodes statiques (version 2)

Code Source 3.8 : /pattern-proto/ex05-createModuleMetierProto.js

```

1  /** @description Définit les propriétés générale des objets métiers
2   * à partir d'une spécification des attributs.
3   * On ajoute au métier un objet qui est l'interface publique d'une fonction qui
4   *   suit le pattern "module".
5   * La fonction retourne son interface publique qui est un objet.
6   * Cet objet est ajouté comme sous-module au module "metier".
7   *
8   * Dans cet objet, on ne trouve pas pour le moment les propriétés d'instance.
9   *   Celles-ci seront ajoutées par "augmentation".
10  *
11  * @module createModuleMetier
12  * @augments myApp.metier
13  * @param {string} moduleName - nom du module métier
14  * @param {Object} attributesPatterns - objet contenant les attributs d'un
15  *   module métier
16  *   - Chaque propriété de attributesPatterns définit un attribut
17  *     du module métier
18  *   - chaque attribut possède
19  *     + une fonction de test sur les valeurs nommée regexTest
20  *     + un texte de label permettant de désigner la donnée
21  *   pour l'utilisateur
22  */
23 myApp.addModule.apply(myApp.metier, [ "createModuleMetier",
24                                     function(moduleName,
25                                             attributesPatterns){
26     ///////////////////////////////////////////
27     // Si le modèle de toutes les méthodes statiques communes
28     // à tous les objets métier n'a pas encore été créé
29     // (Cet objet existe en un exemplaire unique.)
30     if (myApp.metier.methodesStatiquesModulesMetier === undefined){
31         //On crée le modèle de toutes les méthodes statiques communes
32         // à tous les objets métier.
33         myApp.metier.methodesStatiquesModulesMetier = {
34             /** @description Renvoie la liste des propriétés attendues des instances
35              *
36              * @method getAttributeList
37              */
38             getAttributeList : function(){
39                 return attributeList;
40             },
41         },

```



```

36      /** @description Renvoie le texte de description de l'attribut attendue
37      des instances.
38      * Renvoie undefined en cas d'erreur (propriété inconnue)
39      * @method getLabelText
40      * @param {string} attributeName – nom de propriété
41      * @return {string} le texte de description courte du champs
42      */
43      getLabelText : function(attributeName){
44          return attributesPatterns[attributeName].labelText ;
45      },
46
47      /** @description Expose le test d'expression régulière des attributs des
48      instances.
49      * Peut être utilisée pour le filtrage des données d'un formulaire.
50      * @method testRegex
51      * @param {string} attributeName – nom de propriété
52      * @param {string} value – valeur pour initialiser l'attribut
53      * @return {boolean/string} true si la chaîne est un attribut valide,
54      * un message d'erreur sinon.
55      */
56      testRegex : function(attributeName, value){
57          if (attributesPatterns[attributeName] === undefined){
58              return "La propriété " + attributeName + " n'existe pas";
59          } else{
60              return attributesPatterns[attributeName].regexTest(value);
61          }
62      }; // Fin du littéral myApp.metier.methodesStatiquesModulesMetier
63  } // fin du "if (myApp.metier.methodesStatiquesModulesMetier === undefined)"
64
65  var ModuleMetier = function(){};
66
67  // Le modèle de toutes les méthodes statiques communes à tous les objets
68  // métier sert de prototype à tous les modules métier.
69  // Cela permet de rendre ces méthodes (exemplaire unique) accessibles
70  // dans tous les modules métier
71  ModuleMetier.prototype
72      = myApp.metier.methodesStatiquesModulesMetier ;
73
74  // Création du module avec le constructeur de modules métiers
75  // pout pouvoir définir un prototype commun pour tous
76  // les modules métier.
77  myApp.metier[moduleName] = new ModuleMetier() ;
78  /**
79  * Tableau contenant la liste des attributs d'une instance.
80  * Le tableau est précalculé lors de l'initialisation.
81  * @member
82  * @private
83  */
84  var attributeList = function() {
85      var liste = [] ;
86
87      // Parcours des propriétés de l'objet attributesPatterns.regexTest
88      // qui correspondent aux attributs de l'instance
89      for (var attributeName in attributesPatterns){

```

```

90      // Ne pas considérer les propriétés "héritées" du prototype.
91      if (attributesPatterns.hasOwnProperty(attributeName)){
92          liste.push(attributeName);
93      }
94  }
95  return liste;
96 }(); // appel immédiat de la fonction anonyme.
97
98 /** @description Propriété contenant les spécifications
99  * de chaque attribut telles que données en paramètres
100  */
101 //myApp.metier[moduleName].attributesPatterns = attributesPatterns;
102
103 } // fin de la fonction qui crée l'objet myApp.metier[moduleName]
104 ); // fin de l'appel "apply" de la méthode myApp.addModule
105 // (ajout de l'objet this au metier, sous le nom createModuleMetier)

```

### 3.3.2 Fabrique générique d'instances métier (version 2)

Pour les méthodes d'instance, le code source des méthodes étant indépendant de l'instance (même si le résultats de ces méthodes dépend des valeurs attributs, et donc de l'instance), il sera factorisé dans un prototype unique, qui est créé une seule fois. Le constructeur correspondant est mémorisé dans la propriété `privateInstanceConstructor` du module `myApp.metier`.

Comme les attributs d'un constructeur sont toujours publics, nous instancions avec notre `privateInstanceConstructor` une instance qui sera elle-même privée (en suivant le *pattern module* comme dans la partie 2.2).

Le module métier (comme par exemple `myApp.metier.adresse`), dont l'interface contient les méthodes statiques (méthodes de classe), sera utilisé comme prototype des instances (`PublicInstanceInterface`). Une instance du constructeur `PublicInstanceInterface` est retournée comme interface du *pattern module* exposant à la fois les méthodes statiques et les méthodes d'instance.

Code Source 3.9 : `/pattern-proto/ex05-fabriqueObjetMetierProto.js`

```

1  /**
2   * Fabrique qui crée des objets représentant des instance d'objets métier,
3   * suivant le "pattern module".
4   * Le paramètre spec de notre fonction est un objet contenant les attributs de l'
5   * instance à créer.
6   *
7   * @method createInstanceGenerique
8   * @augments myApp.metier
9   * @param {Object|null} inputObj - spécification des attributs d'une instance d'
10  * objet métier.
11  * Si inputObj est null, on crée une instance par défaut (id aléatoire, autres
12  * attributs vide).
13  * @param {string||undefined} inputObj.id - identifiant unique de l'instance
14  * @param {string|number} inputObj.attributeName - valeur à affecter à l'
15  * attribut attributeName
16  * (exemple pour une adresse : inputObj.numeroRue, inputObj.
17  * codePostal, etc.
18  */
19 myApp.addModule.apply(myApp.metier, [ "createInstanceGenerique", function(
20     inputObj){
21

```

```
15  that = this ;
16
17  //////////////////////////////////////////
18  // Si le mod le de toutes les m thodes priv es communes
19  //   toutes les instances d'objets m tier n'a pas encore  t  cr  
20  // (Cet objet existe en un exemplaire unique.)
21  if (myApp.metier.methodesInstancesMetier === undefined){
22    // Constructeur d'un objet unique avec un prototype
23    var PrivateInstancesConstructor = function() {
24      /** @description contient les valeurs des attributs de l'instance,
25          * @member
26          * @private
27          */
28      this.dataAttributes = {};
29      /** @description contient les messages d'erreur associ s aux attributs
30          * @member
31          * @private
32          */
33      this.dataError = {};
34    };
35
36    PrivateInstancesConstructor.prototype.addError
37    = function(attributeName, message){
38      // Ajout d'une propri t 
39      this.dataError[attributeName] = message;
40    };
41
42    PrivateInstancesConstructor.prototype.setAttribute
43    = function(attributeName, value){
44      var resultTestRegex = that.testRegex(attributeName, value);
45      // On initialise l'attribut de l'instance
46      this.dataAttributes[attributeName] = value;
47      // Si la validation par expression r guli re est pass e
48      if (resultTestRegex === true){
49        // On efface une vieille erreur  ventuelle
50        delete this.dataError[attributeName];
51      }else{
52        // On initialise la propri t  de l'objet des erreurs.
53        // avec le message d'erreur.
54        this.addError(attributeName, "Attribut " + value +
55          " invalide : " + resultTestRegex);
56      }
57    };
58
59    PrivateInstancesConstructor.prototype.getAttribute
60    = function(attributeName){
61      return this.dataAttributes[attributeName];
62    };
63
64    PrivateInstancesConstructor.prototype.hasError
65    = function() {
66      for (var attributeName in this.dataError){
67        if (this.dataError.hasOwnProperty(attributeName)){
68          return true;
```

```

69     }
70   }
71   return false ;
72 };
73
74 PrivateInstancesConstructor.prototype.getErrorList
75 = function(){
76   var errorList = [];
77   for (var attributeName in this.dataError){
78     if (this.dataError.hasOwnProperty(attributeName)){
79       errorList.push(attributeName);
80     }
81   }
82   return errorList ;
83 };
84
85 PrivateInstancesConstructor.prototype.getErrorMessage
86 = function(attributeName){
87   return this.dataError[attributeName] ;
88 };
89 /** @description Génération d'un ID aléatoire en cas de création d'une
90     nouvelle instance
91  * (cas où les spécifications inputObj sont null)
92  * @private
93  */
94 PrivateInstancesConstructor.prototype.generateRandomId
95 = function(){
96   var idLength = 10;
97   var resultat = "";
98   var hexaDigits = Array("0","1","2","3","4","5","6","7","8","9","a","b",
99     "c","d","e","f");
100   var i;
101   for (var i=0 ; i<10 ; ++i){
102     resultat += hexaDigits[Math.floor(Math.random()*16)];
103   }
104   return resultat ;
105 }
106
107 // On mémorise le constructeur d'instances privées pour n'en avoir qu'un
108 myApp.metier.privateInstanceConstructor = PrivateInstancesConstructor ;
109 } // fin du "if (myApp.metier.methodesInstancesMetier === undefined)"
110
111 // On crée une instance privée (variable locale, voir "pattern module")
112 var privateInstance = new myApp.metier.privateInstanceConstructor() ;
113
114 /** @description Création d'un constructeur privé, créant une classe qui
115  * contiendra les méthodes permettant de manipuler l'instance créée,
116  * qui constitue l'interface du pattern module.
117  * L'intérêt d'un constructeur est de définir les méthode au niveau du
118  * prototype,
119  * via l'objet unique myApp.metier.methodesInstancesMetier, créé une seule
120  * fois.
121  * De plus, le prototype des instances hérite lui-même des méthodes

```

```

121     statiques
122     * du module métier.
123     */
124     var PublicInstanceInterface = function() {};
125
126     // MISE À DISPOSITION DES MÉTHODES STATIQUES DU MODULE
127     // VIA LE PROTOTYPE
128     PublicInstanceInterface.prototype = this; // exemple : this = myApp.métier.
129     adresse
130
131     /** @description Retourne le module avec les méthodes "statiques"
132     * (comme l'accès direct à la liste des attributs ou les tests regex)
133     * @return {Object} le module myApp.métier.moduleName
134     */
135     PublicInstanceInterface.prototype.getModule = function() {
136         return this;
137     };
138
139     // Interface publique du module, retournée par la fabrique
140     var instanceInterface = new PublicInstanceInterface();
141
142     instanceInterface.getModule = function() {
143         return this;
144     };
145
146     ////////////////////////////////////
147     // On expose les méthodes du prototype de l'instance privée
148     // via l'interface des instances :
149
150     /** @description Ajoute une propriété (message d'erreur) dans this.dataError
151     * correspondant à un attribut.
152     * @method addError
153     * @public
154     */
155     instanceInterface.addError = function() {
156         privateInstance.addError();
157     };
158
159     /** @description Setter : initialise la valeur des attributs d'une instance.
160     * En cas d'erreur un message pour cette propriété est ajouté dans this.
161     dataError.
162     * En l'absence d'erreur, une éventuelle erreur précédente est effacée.
163     * @method setAttribute
164     * @public
165     */
166     instanceInterface.setAttribute = function(attributeName, value) {
167         privateInstance.setAttribute(attributeName, value);
168     };
169
170     /** @description Accesseur pour tous les membres privés d'instance.
171     * @method getAttribute
172     * @public
173     * @param {string} attributeName - nom de l'attribut attendue d'une
174     instance
175     * @return {string} la valeur de l'attribut ou undefined en cas de nom d'

```

```

    attribut inconnu.
*/
173 instanceInterface.getAttribute = function(attributeName){
174     return privateInstance.getAttribute(attributeName);
175 };
176
177
178 /** @description Détermine si l'instance comprend au moins une erreur.
179  * @public
180  * @return {boolean} true s'il y a (au moins) une erreur, false sinon
181  */
182 instanceInterface.hasError = function(attributeName){
183     return privateInstance.hasError(attributeName);
184 };
185
186 /** @description Récupère la liste des champs qui ont une erreur
187  * @public
188  * @return {string[]} tableau des noms de propriétés qui comportent une
    erreur.
189  */
190 instanceInterface.getErrorList = function() {
191     return privateInstance.getErrorList();
192 };
193
194 /** @description Donne l'accès au message d'erreur d'un attribut (s'il existe
    ).
195  * @method getErrorMessage
196  * @public
197  * @param {string} attributeName - nom d'attribut d'une instance de module mé
    tier
198  * @return {string|undefined} le message d'erreur pour un attribut s'il
    existe
199  *
200  * ou undefined en l'absence d'erreur
201  */
202 instanceInterface.getErrorMessage = function(attributeName){
203     return privateInstance.getErrorMessage(attributeName);
204 };
205
206 // Initialisation des attributs de l'instance
207 // Si l'objet en argument est null, on construit une instance par défaut
208 if (inputObj !== null){
209     // Parcours des propriétés de getAttributeList()
210     // qui correspondent aux attributs de l'instance à créer
211     for (var i = 0 ; i < this.getAttributeList().length ; ++i){
212         var attributeName = this.getAttributeList()[i];
213         // Utilisation du setter pour initialiser l'attribut
214         privateInstance.setAttribute(attributeName, inputObj[attributeName]);
215     }
216 } else{
217     // On initialise les valeurs des attributs à ""
218     for (var i = 0 ; i < this.getAttributeList().length ; ++i){
219         var attributeName = this.getAttributeList()[i];
220         if (attributeName === "id"){
221             privateInstance.dataAttributes[attributeName] = privateInstance.
                generateRandomId();
222         } else{
                privateInstance.dataAttributes[attributeName] = "";

```

```

223     }
224   }
225 }
226
227 // Construction d'une instance avec
228 return instanceInterface;
229 } // fin de la méthode createInstance
230 ]); // fin de l'appel "apply" de la méthode myApp.addModule

```

### 3.3.3 Utilisation avec l'affichage générique d'objets métier

Les méthodes de test d'implémentation d'interfaces de la partie 2.7 sont utilisables telles quelles. Pour adapter l'exemple de test d'affichage générique (code source 2.31), il suffit de modifier l'inclusion des fichiers *JavaScript* comme suit :

Code Source 3.10 : Fichiers *JS* inclus dans `ex06-fabriqueAdresseProtoTest.html`

```

1  <!-- Création de l'application vide avec deux méthodes -->
2  <script src="../pattern-fonct/ex04-structureApplication.js"></script>
3  <!-- Création de sous-module regexUtil de myApp.metier -->
4  <script src="../pattern-fonct/ex05-modulePatternRegex.js"></script>
5  <!-- Création de sous-module adresse de myApp.metier -->
6  <script src="../ex05-createModuleMetierProto.js"></script>
7  <!-- Création d'une méthode fabrique générique d'objets métier -->
8  <script src="../ex05-fabriqueObjetMetierProto.js"></script>
9  <!-- Création de sous-module adresse de myApp.metier -->
10 <script src="../pattern-fonct/ex06-moduleMetierAdresse.js"></script>
11 <!-- Création d'une méthode fabrique d'adresse de myApp.metier.adresse -->
12 <script src="../pattern-fonct/ex08-fabriqueAdresse.js"></script>
13 <!-- Classe de vérification de l'implémentation d'interfaces -->
14 <script src="../pattern-fonct/ex11-interfaceImplementation.js"></script>
15 <!-- Classe de vérification de l'implémentation d'interfaces -->
16 <script src="../pattern-fonct/ex11-interfaceImplementationMetier.js"></script>
17 <!-- Création de fonctions d'affichage générique dans myApp.metier.view -->
18 <script src="../pattern-fonct/ex12-objetMetierView.js"></script>
19
20 <!-- Ajout d'une fonction de test, d'une méthode "main", et exécution -->
21 <script src="../pattern-fonct/ex12-objetMetierViewTest.js"></script>

```

Les fichiers du répertoire `../pattern-fonct` correspondent aux exemples définis dans le chapitre 2, qui sont réutilisables tels quels du fait de la généricité et de l'homogénéité des interfaces d'objets métier.

## 3.4 Patterns *pseudo-classique* (à éviter)

Dans l'exemple suivant, nous créons une classe `Personne` qui hérite des propriétés de la classe `Adresse`. Pour cela :

1. le constructeur d'`Adresse` est appelé explicitement dans le constructeur de `Personne` ;
2. la classe `Adresse` est déclarée comme `superclass` de la classe `Personne` ;

3. Les méthodes qui existent au niveau du prototype de la classe **Adresse** et qui doivent être spécifiées pour des personnes sont surchargées au niveau du prototype de la classe **Personne**.

Dans l'exemple suivant, nous surchargeons l'accesseur de la propriété `ville` et la méthode `toString`.

Code Source 3.11 : /pattern-proto/ex08-extension-de-classe.js

```

1  function Adresse(numeroRue, rue, complement, codePostal, ville) {
2    if (numeroRue.match(/^([0-9]*)([0-9]+)(\ ?)((bis)/(ter)))?$/)) {
3      this.numeroRue = numeroRue.replace(/\s+/g, ' ');
4    } else {
5      throw new Error("Numéro de la rue invalide.");
6    }
7
8    if (rue.match(/^(([a-zA-ZêéèöàöÉÊËÄÖË\ | - \.,0-9\ ])/(\ "))/(\ ')){1,300}$/))
9      {
10     this.rue = rue.replace(/\s+/g, ' ');
11   } else {
12     throw new Error("Nom de la rue/place invalide.");
13   }
14
15   if (complement.match(/^(([a-zA-ZêéèöàöÉÊËÄÖË\ | - \.,0-9\ ])/(\ "))/(\ '))
16     {0,300}$/)) {
17     this.complement = complement.replace(/\s+/g, ' ');
18   } else {
19     throw new Error("Complement d'adresse invalide.");
20   }
21
22   if (codePostal.match(/^([0-9]{5})$/)) {
23     this.codePostal = codePostal;
24   } else {
25     throw new Error("Code postal invalide.");
26   }
27
28   if (ville.match(/^(([a-zA-ZêéèöàöÉÊËÄÖË\ | - \.,0-9\ ])/(\ "))/(\ ')){0,300}$/))
29     {
30     this.ville = ville.replace(/\s+/g, ' ');
31   } else {
32     throw new Error("Nom de ville invalide.");
33   }
34
35   Adresse.prototype.getVille = function() {
36     return this.ville;
37   }
38
39   Adresse.prototype.toString = function() {
40     var resultat = this.numeroRue;
41     if (this.numeroRue != "")
42       resultat += ", ";
43     resultat += this.rue + ", ";
44     resultat += this.complement;
45     if (this.complement != "")
46       resultat += ", ";

```



```
46     resultat += this.ville + "<br/>";
47     return resultat;
48 }
49
50 function Personne(nom, prenom, numeroRue, rue, complement, codePostal, ville){
51     Adresse.call(this, numeroRue, rue, complement, codePostal, ville);
52     this.nom = nom;
53     this.prenom = prenom;
54 }
55
56 Personne.superclass = Adresse;
57 Personne.prototype.getVille = function() {
58     return Adresse.prototype.getVille.call(this);
59 }
60
61 Personne.prototype.toString = function() {
62     return this.nom + ", " + this.prenom + ", " + Adresse.prototype.toString.call(this);
63 }
```

Code Source 3.12 : /pattern-proto/ex08-extension-de-classe.html

```
1 <!doctype HTML>
2 <html lang="fr">
3 <head>
4 <meta charset="UTF-8" />
5 <title>Chainage de constructeurs</title>
6 <script src="/ex08-extension-de-classe.js"></script>
7 </head>
8 <body>
9 <p>
10 <script>
11 try{
12     var pers = new Personne("Dujardin", "Jean", "10 ter", "rue de l'avenir", "Le
13         Rastou", "86098", "Les Flots Bleus");
14     document.write(pers);
15     document.write("<br/>L'adresse se trouve dans la ville de \""+pers.getVille()+
16         "\" \".");
17 }catch (err){
18     alert(err);
19 }
20 </script>
21 <p>
22 </body>
23 </html>
```

# Chapitre 4

## Formulaires, Filtrage, *Pattern Mediator*

### 4.1 Filtrage Basique des Inputs d'un Formulaire

L'exemple suivant montre comment filtrer les attributs d'un formulaires côté client en affichant immédiatement un message d'erreur lors de la saisie d'une valeur incorrecte. On associe à chaque événement `onchange` de chaque attribut une fonction *JavaScript* qui réalisera le filtrage.

Code Source 4.1 : /form-filter/ex01-basicForm.js

```
1 // alias vers le module d'expressions régulières
2 var regexUtil = myApp.metier.regexUtil;
3
4 /** @description Gestionnaire d'événement change de l'input d'ID "mainForm_titre
5     ".
6     * Cette méthode effectue le filtrage par exexpression régulière.
7     * @method filterTitre
8     */
9 var filterTitre = function(){
10     var titreValue = $("#mainForm_titre").val();
11     // Expressions du langage courant et chiffres
12     var resultRegexTest = regexUtil.testRegexLatin1WithDigits({
13         chaine : titreValue,
14         minLength : 1
15     });
16     // Modification du contenu du span d'ID "error_mainForm_titre"
17     if (resultRegexTest === true){
18         $("#error_mainForm_titre").empty();
19     }else{
20         $("#error_mainForm_titre").html(
21             "Erreur : le titre ne doit contenir que les lettres et chiffres<br/>");
22     }
23 };
24
25 /** @description Gestionnaire d'événement onchange de l'input d'ID "
26     mainForm_resume".
27     * Cette méthode effectue le filtrage par exexpression régulière.
28     * @function filterResume
29     */
30 var filterResume = function(){
```

```

29  var titreValue = $("#mainForm_resume").val();
30  // Expressions du langage courant et chiffres et ponctuation
31  var resultRegexTest = regexUtil.testRegexLatin1WithDigitsPunctuation({
32      chaine : titreValue,
33      minLength : 1
34  });
35  // Modification du contenu du span d'ID "error_mainForm_resume"
36  if (resultRegexTest === true){
37      $("#error_mainForm_resume").empty();
38  }else{
39      $("#error_mainForm_resume").html(
40          "Erreur : le résumé ne doit contenir que les lettres et chiffres" +
41          " ou des caractères de ponctuation<br/>");
42  }
43  };

```



FIGURE 4.1 : Illustration du code source 4.2

Code Source 4.2 : /form-filter/ex01-basicFormTest.html (cf. Fig 4.1)

```

1  <!doctype HTML>
2  <html lang="fr">
3  <head>
4  <meta charset="UTF-8" />
5  <title>Filtrage d'inputs</title>
6  <link rel="stylesheet" href="basicStyle.css"/>
7  </head>
8  <body>
9  <h1>Saisie d'un film</h1>
10 <form id="mainForm" action="post">
11   <!-- input avec gestionnaire de l'événement onchange -->
12   <span id="error_mainForm_titre" class="errorMsg"></span>
13   <label for="mainForm_titre">Titre :</label>
14   <input type="text" id="mainForm_titre" size="15"
15   placeholder="Titre du film" onchange="filterTitre()"><br/>
16
17   <!-- textarea avec gestionnaire de l'événement onchange -->
18   <span id="error_mainForm_resume" class="errorMsg"></span>
19   <label for="mainForm_resume">Résumé :</label>

```

```

20     <textarea id="mainForm_resume" rows="10" cols="50"
21     placeholder="Saisissez votre résumé" onchange="filterResume()"></textarea>
22 </form>
23 <!-- Inclusion de la structure d'application et du module regexUtil -->
24 <script src="modulesMetier.js"></script>
25 <!-- Pattern Mediator spécialisé pour les événement "change" des <input> -->
26 <script src="ex02-mediatorInputFilter.js"></script>
27 <!-- Inclusion de jQuery pour les événements et manipulation du DOM -->
28 <script src="jquery.js"></script> <script src="ex01-basicForm.js"></script>
29 </body>
30 </html>

```

## 4.2 *Pattern Mediator* pour le filtrage d'attributs

L'inconvénient du filtrage présenté dans la partie 4.1 est que, dans le code *HTML* d'un champs du formulaire lui-même, on doit déclarer une méthode de filtrage spécifique pour ce champs (attribut `onchange` de l'élément *HTML* `input` ou `textarea`).

Dans l'architecture d'application que nous proposons par la suite, la méthode de filtrage ne sera pas codée en dût dans le module chargé de générer le formulaire, mais plutôt dans les tests d'expressions régulières effectués dans les modules métier. En particulier, la méthode précise dépendra de l'instance et de l'attribut considéré, ce qui entraînerait un fort *couplage* (interdépendance) des méthodes chargées de l'*IHM* et des classes métier (ou du *modèle*).

Nous savons par expérience que ce type de couplage va provoquer des difficultés pour la maintenance et l'évolution de notre application (comme par exemple la migration de nos objet métier côté serveur avec *NodeJS*). Nous allons maintenant introduire un *pattern* qui a pour vocation de découpler le déclenchement des événements (via, en l'occurrence, des événements utilisateurs `onchange`) de l'implémentation des opérations correspondantes sur les données métier, ou les données du modèle. Ce pattern est une généralisation du pattern *Observer*.

Dans notre exemple, un module *Mediator* va enregistrer les méthodes *callbacks* (qui ne sont que des fonctions *JavaScript*) associées à des événements. L'exécution des ces callbacks (en l'occurrence la réaction à un événement `onchange`) sera déclenchée par la publication de l'événement en question par l'intermédiaire du *Mediator*.

Code Source 4.3 : `/form-filter/ex02-mediatorInputFilter.js`

```

1  /**
2  *
3  * Ajout d'un module ctrl (contrôleurs) à l'application.
4  * @module ctrl
5  * @augments myApp
6  */
7  myApp.addModule.apply(myApp, [ "ctrl", {} ] );
8
9  /**
10 * Implémentation du pattern "Médiateur" pour gérer le filtrage des inputs de
11 *   formulaires.
12 * @module mediatorInputFilter
13 * @augments myApp.ctrl
14 */
15 myApp.addModule.apply(myApp.ctrl, [ "mediatorInputFilter", function() {
16     //////////////////////////////////////

```

```

17 // Propriétés et méthodes "statiques" privées
18
19 /**
20  * Collection, indexée par ID de formulaire de callbacks d'événements liés à
21  * différents formulaires (typiquement : événement onchange d'un input).
22  * @private
23  */
24 var m_subscriptionLists;
25
26 /**
27  * Initialise (ou réinitialise) l'ensemble des listes d'événements à la
28  * collection vide.
29  * @private
30  */
31 var init = function() {
32     m_subscriptionLists = {};
33 };
34
35 // Initialiser une fois l'ensemble des listes d'événements à la collection
36 // vide.
37 init();
38
39 ///////////////////////////////////////////////////
40 // Interface publique du module
41
42 /**
43  * Création d'un objet contenant les données et méthodes publiques
44  * (les propriétés publiques sont retournées par la fonction "module").
45  */
46 var publicInterfaceMediator = {
47
48     /**
49      * Ajoute un formulaire et la liste (initialement vide) de ses callbacks
50      * associés.
51      * Si le formulaire est déjà géré, la liste de ses callbacks associés est
52      * supprimée et réinitialisée à la liste vide.
53      * @param {string} formId - l'Id du formulaire (en tant qu'élément HTML)
54      */
55     addForm : function(formId) {
56         m_subscriptionLists[formId] = {};
57     },
58
59     /**
60      * Supprime un formulaire et ses callbacks associés
61      * @param {string} formId - l'Id du formulaire (en tant qu'élément HTML)
62      */
63     removeForm : function(formId) {
64         if (!m_subscriptionLists.hasOwnProperty(formId)) {
65             return false;
66         }
67         delete m_subscriptionLists[formId];
68         return true;
69     },
70
71     /**
72      * Ajout d'un événement associé à un attribut de formulaire et de sa

```

```

    fonction callback.
68  * Si l'événement existait déjà pour cet input, il est écrasé.
69  * @param {string} formId - l'Id du formulaire (en tant qu'élément HTML)
70  * @param {string} inputName - le nom de l'input (ou de la propriété de l'
    objet métier associé).
71  * @param {function} callbackFunction - la fonction (callback) à appeler en
    cas de publication de l'événement.
72  */
73  subscribe : function(formId, inputName, callbackFunction){
74      if (m_subscriptionLists.hasOwnProperty(formId)){
75          m_subscriptionLists[formId][inputName] = {callback : callbackFunction};
76      }else{
77          throw {name: "IllegalArgumentException",
78                message: "Catégorie d'événements " + eventCateg + " inconnue du mé
                    diateur"}
79      };
80  }
81  },
82
83  /**
84  * Publication d'un événement associé à un attribut de formulaire provoquant
    l'exécution de la fonction callback associée
85  * @param {string} formId - l'Id du formulaire (en tant qu'élément HTML)
86  * @param {string} inputName - le nom de l'input (ou de la propriété de l'
    objet métier associé).
87  */
88  publish : function(formId, inputName){
89
90      if (m_subscriptionLists.hasOwnProperty(formId)){
91          if (m_subscriptionLists[formId].hasOwnProperty(inputName)){
92              // On appelle le callback avec son
93              m_subscriptionLists[formId][inputName].callback();
94          }
95      }else{
96          throw {name: "IllegalArgumentException",
97                message: "Formulaire d' ID " + formId + " inconnu du médiateur"}
98      };
99  }
100 },
101
102 /**
103 * Réinitialise la collection des formulaires gérés à une collection vide.
104 */
105 empty : function(){
106     init();
107 }
108 };
109
110 return publicInterfaceMediator;
111
112 }());

```

Code Source 4.4 : /form-filter/ex02-mediatorInputFilterTest.html

```

1 <!doctype HTML>
2 <html lang="fr">

```

```

3 <head>
4 <meta charset="UTF-8" />
5 <title>Filtrage d'inputs</title>
6 <link rel="stylesheet" href="basicStyle.css"/>
7 </head>
8 <body>
9   <h1>Saisie d'un film</h1>
10  <form id="mainForm" action="post">
11    <!-- input avec gestionnaire de l'événement onchange -->
12    <span id="error_mainForm_titre" class="errorMsg"></span>
13    <label for="mainForm_titre">Titre </label>
14    <input type="text" id="mainForm_titre" size="15"
15    placeholder="Titre du film" onchange="filterData('mainForm', 'titre')"/><br />
16    <!-- textarea avec gestionnaire de l'événement onchange -->
17    <span id="error_mainForm_resume" class="errorMsg"></span>
18    <label for="mainForm_resume">Résumé </label>
19    <textarea id="mainForm_resume" rows="10" cols="50"
20    placeholder="Saisissez votre résumé" onchange="filterData('mainForm', '
    resume')"/></textarea>
21  </form>
22  <!-- Inclusion de la structure d'application et du module regexUtil -->
23  <script src="modulesMetier.js"></script>
24  <!-- Inclusion de jQuery pour les événements et manipulation du DOM -->
25  <script src="jquery.js"></script>
26  <script src="ex02-mediatorInputFilter.js"></script>
27  <script src="ex01-basicForm.js"></script>
28  <script>
29    // Ajout du formulaire "mainForm" au médiateur qui gèrera ses événements.
30    myApp.ctrl.mediatorInputFilter.addForm('mainForm');
31    // Enregistrement du callback associé à l'événement onchange du titre
32    myApp.ctrl.mediatorInputFilter.subscribe('mainForm', 'titre', filterTitre);
33    // Enregistrement du callback associé à l'événement onchange du résumé
34    myApp.ctrl.mediatorInputFilter.subscribe('mainForm', 'resume', filterResume)
35    ;
36    /** @description Publie l'événement onchange d'un input auprès du médiateur,
37     * provoquant l'exécution du callback enregistré pour cet événement.
38     * @function filterData
39     */
40    var filterData = function(formId, inputName){
41      myApp.ctrl.mediatorInputFilter.publish(formId, inputName);
42    };
43  </script>
44 </body>
45 </html>

```

## 4.3 Exemple :

### Génération automatique de formulaire d'adresse

Dans l'exemple suivant, des méthodes d'un module `myApp.gui` permettent de générer automatiquement les inputs d'un formulaire permettant de saisir les attributs (ici supposées de type texte) d'un objet qui implémente des interface qui apparaissent dans l'exemple de la

partie 2.7.1.

Nous appliquons cette méthode pour afficher et filtrer automatiquement un formulaire de saisie d'une adresse.

Code Source 4.5 : /form-filter/ex03-formsGui.js

```

1  /** @description Module "gui" (pour Graphical User Interface)
2   * @module gui
3   * @augments myApp
4   */
5  myApp.addModule.apply(myApp, [ "gui", {} ] );
6
7  /** @description Génération de l'ID d'un élément HTML de type input préfixé par
8   * l'ID du formulaire
9   * @method getInputId
10  * @augments myApp.gui
11  * @param {Object} inputSpec - contient les spécifications de l'input
12  * @param {string} inputSpec.formId - id du formulaire dans lequel l'input sera
13   * inséré
14  * @param {string} inputSpec.attributeName - nom de l'attribut de inputSpec.
15   * objetMetier
16   *
17   * à saisir dans l'input
18  */
19  myApp.addModule.apply(myApp.gui, [ "getInputId", function(inputSpec){
20    return inputSpec.formId + "_" + inputSpec.attributeName;
21  } ] );
22
23  /** @description Publie auprès du Mediator un événement onchange d'un Input
24  * @method publishInputChange
25  * @augments myApp.gui
26  * @param {string} formId id du formulaire dans lequel l'input sera inséré
27  * @param {string} attributeName nom de l'attribut de inputSpec.objetMetier à
28   * saisir dans l'input
29  */
30  myApp.addModule.apply(myApp.gui, [ "publishInputChange", function(formId,
31    attributeName){
32    myApp.ctrl.mediatorInputFilter.publish(formId, attributeName);
33  } ] );
34
35  /** @description Génération du code HTML d'un input.
36  * @method getTextInputCode
37  * @augments myApp.gui
38  * @param {Object} inputSpec contient les spécifications de l'input
39  * @param {Object} inputSpec.instance - instance d'objet métier qui implémente :
40   * "getModule", "getAttribute", "setAttribute",
41   * "hasError", "getErrorMessage", "getErrorMessage"
42   * ainsi que les méthodes statiques :
43   * "getAttributeList", "getLabelText", "testRegex",
44   * "createInstance", "getModule".
45  * @param {string} inputSpec.formId id du formulaire dans lequel l'input sera
46   * inséré
47  * @param {string} inputSpec.attributeName nom de l'attribut de inputSpec.
48   * objetMetier à saisir dans l'input
49  * @param {string} [inputSpec.type=text] type de l'input
50  * @param {number} [inputSpec.inputSize=10] taille de l'input (nombre de
51   * caractères)
52  */

```



```

44 myApp.addModule.apply(myApp.gui, ["getInputCode", function(inputSpec){
45     // Calcul de l'ID de l'input :
46     var inputId = myApp.gui.getInputId(inputSpec);
47
48     // Valeur de l'attribut de l'objet pour l'attribut value de l'input
49     var attributeValue = inputSpec.objetMetier.getAttribute(inputSpec.
        attributeName) || "";
50     // Création d'un éventuel message si l'objet comportait déjà une erreur
51     var errorMessage = inputSpec.objetMetier.getErrorMessage(inputSpec.
        attributeName) !== undefined
52         ? inputSpec.objetMetier.getErrorMessage(inputSpec.attributeName
            ) + "<br/>" : "";
53
54     //////////////////////////////////////
55     // Callback de gestion du filtrage de l'input :
56     myApp.ctrl.mediatorInputFilter.subscribe(inputSpec.formId, inputSpec.
        attributeName, function(){
57
58         var resultatTestRegex = inputSpec.objetMetier.testRegex(inputSpec.
            attributeName,
59             document.getElementById(inputId).value);
60         if (resultatTestRegex !== true){
61             document.getElementById("error_"+inputId).innerHTML = resultatTestRegex
                + "<br/>";
62         }else{
63             document.getElementById("error_"+inputId).innerHTML = "";
64         }
65     }); // fin du callback //////////////////////////////////////
66
67     var inputType = inputSpec.inputType === undefined ? "text" : inputSpec.
        inputType;
68     var inputSize = inputSpec.inputSize === undefined ? "10" : inputSpec.inputSize
        ;
69     var labelText = inputSpec.objetMetier.getLabelText(inputSpec.attributeName);
70
71     // retour du code HTML de l'input
72     return "<span class=\"errorMsg\" id=\"error_"+inputId+"\">" + errorMessage + "
        </span>" +
73         "<label for=\"" + inputSpec.attributeName + "\">" + labelText + "</label>"
            +
74         "<input type=\"" + inputType + "\" name=\"" + inputSpec.
            attributeName +
75             "\" id=\"" + inputId + "\" value=\"" + attributeValue + "\" " +
76             "size=\"" + inputSize + "\" " +
77             "onchange=\"myApp.gui.publishInputChange(' + inputSpec.formId + '", '
                + inputSpec.attributeName + "')\">";
78 });
79
80 /** @description Génération du code HTML de l'ensemble des inputs d'un
    formulaire.
81 *
82 * @method getHtmlFormInputs
83 * @augments myApp.gui
84 * @public
85 * @param {Object} instance - instance d'objet métier qui implémente :
86 *                             "getModule", "getAttribute", "setAttribute",

```

```

87  *          "hasError", "getErrorMessage", "getErrorList"
88  *          ainsi que les méthodes statiques :
89  *          "getAttributeList", "getLabelText", "testRegex",
90  *          "createInstance", "getModule".
91  * @param {string} formId id du formulaire dans lequel l'input sera inséré
92  * @return {string} le code HTML des tous les inputs correspondant aux propriétés
93  *                  s de l'objet métier.
94  */
95 myApp.addModule.apply(myApp.gui, ["getHtmlFormInputs", function(objetMetier,
96   formId){
97   // On vérifie que l'objetMetier implémente bien notre interface générique.
98   var testInterfacesMetier = myApp.metier.testInterfaceImplementation(
99     objetMetier);
100   if (testInterfacesMetier !== true){
101     throw new Error(testInterfacesMetier);
102   }
103
104   // Ajout du formulaire "mainForm" au médiateur qui gèrera ses événements.
105   myApp.ctrl.mediatorInputFilter.addForm(formId);
106
107   var htmlCode = "";
108
109   var attributeList = objetMetier.getAttributeList();
110
111   // Tous les inputs sont de type texte, donc on peut
112   // faire une boucle automatique sur les propriétés.
113   for (var i=0 ; i < attributeList.length ; i++){
114     var attributeName = attributeList[i];
115     // l'utilisateur ne peut pas modifier l'ID :
116     if (attributeName !== "id"){
117       // Concaténation du code HTML de l'input
118       htmlCode += myApp.gui.getInputCode({
119         objetMetier : objetMetier,
120         attributeName : attributeList[i],
121         labelText : objetMetier.getLabelText(attributeList[i]),
122         formId : formId
123       }) + "<br/>";
124     }
125   }
126
127   // champs caché représentant l'ID de l'instance
128   htmlCode += "<input type='hidden' id='" + formId + "_id' value='" +
129     objetMetier.getAttribute("id") + "'/>";
130
131   return htmlCode;
132 }]);

```

Code Source 4.6 : /form-filter/ex03-formsGuiTest.js (cf. Fig 4.2)

```

1 // Ajout d'une méthode mainFunction
2 myApp.addModule("mainFunction", function(){
3
4   // création d'une instance
5   var adresse = myApp.metier.adresse.createInstance({
6     id : "04abf85bc9",
7     numeroRue : "2 bis@",

```

FIGURE 4.2 : Illustration du code source 4.6

```

8   rue : "Rue de l'a Paix",
9   complementAddr : "Bâtiment 3D",
10  codePostal : "63000",
11  ville : "Clermont-Ferrand",
12  pays : "France"
13  });
14
15  // Génération du formulaire avec les callbacks
16  codeHTML = "<form id=\"mainForm\" method=\"post\">" +
17             myApp.gui.getHtmlFormInputs(adresse, "mainForm") +
18             "<input type=\"submit\" value=\"valider\"/>" +
19             "</form>";
20  // Utilisation de la valeur retournée pour générer la vue
21  document.getElementById("paragrapheResultat").innerHTML = codeHTML;
22  });
23
24  // Exécution de la méthode mainFunction
25  myApp.mainFunction();

```

Code Source 4.7 : Fichiers *JS* inclus dans ex03-formsGuiTest.html

```

1  <!-- Structure de l'application vide avec deux méthodes -->
2  <script src="../pattern-fonct/ex04-structureApplication.js"></script>
3  <!-- Création de sous-module regexUtil de myApp.metier -->
4  <script src="../pattern-fonct/ex05-modulePatternRegex.js"></script>
5  <!-- Sous-module adresse de myApp.metier -->
6  <script src="../pattern-fonct/ex06-createModuleMetier.js"></script>
7  <!-- Méthode fabrique générique d'objets métier -->
8  <script src="../pattern-fonct/ex07-fabriqueObjetMetier.js"></script>
9  <!-- Création de sous-module adresse de myApp.metier -->
10 <script src="../pattern-fonct/ex06-moduleMetierAdresse.js"></script>
11 <!-- Création d'une méthode fabrique d'adresse de myApp.metier.adresse -->
12 <script src="../pattern-fonct/ex08-fabriqueAdresse.js"></script>
13 <!-- Classe de vérification de l'implémentation d'interfaces -->
14 <script src="../pattern-fonct/ex11-interfaceImplementation.js"></script>
15 <!-- Classe de vérification de l'implémentation d'interfaces -->
16 <script src="../pattern-fonct/ex11-interfaceImplementationMetier.js"></script>
17 <!-- Fonctions d'affichage générique dans myApp.metier.view -->
18 <script src="../pattern-fonct/ex12-objetMetierView.js"></script>
19 --

```

```

20 <!-- Inclusion de jQuery pour les événements et manipulation du DOM -->
21 <script src="jquery.js"></script>
22 <!-- Mediator spécialisé pour filtrer les inputs (evt "change") -->
23 <script src="ex02-mediatorInputFilter.js"></script>
24 <!-- Génération automatique de formulaires avec filtrage des attributs -->
25 <script src="ex03-formsGui.js"></script>
26 --
27 <!-- Ajout d'une fonction de test, d'une méthode "main", et exécution -->
28 <script src="ex03-formsGuiTest.js"></script>

```

Le test avec la modélisation des objets métiers de la partie 3.3, utilisant la notion de prototype, diffère seulement au niveau de l'inclusion de la définition des modules métier dans le fichier *HTML* :

Code Source 4.8 : Fichiers *JS* inclus dans *ex03-formsGuiTestProto.html*

```

1 <!-- Structure de l'application vide avec deux méthodes -->
2 <script src="../pattern-fonct/ex04-structureApplication.js"></script>
3 <!-- Création de sous-module regexUtil de myApp.metier -->
4 <script src="../pattern-fonct/ex05-modulePatternRegex.js"></script>
5 <!-- Sous-module adresse de myApp.metier -->
6 <script src="../pattern-proto/ex05-createModuleMetierProto.js"></script>
7 <!-- Méthode fabrique générique d'objets métier -->
8 <script src="../pattern-proto/ex05-fabriqueObjetMetierProto.js"></script>
9 <!-- Création de sous-module adresse de myApp.metier -->
10 <script src="../pattern-fonct/ex06-moduleMetierAdresse.js"></script>
11 <!-- Création d'une méthode fabrique d'adresse de myApp.metier.adresse -->
12 <script src="../pattern-fonct/ex08-fabriqueAdresse.js"></script>
13 <!-- Classe de vérification de l'implémentation d'interfaces -->
14 <script src="../pattern-fonct/ex11-interfaceImplementation.js"></script>
15 <!-- Classe de vérification de l'implémentation d'interfaces -->
16 <script src="../pattern-fonct/ex11-interfaceImplementationMetier.js"></script>
17 <!-- Fonctions d'affichage générique dans myApp.metier.view -->
18 <script src="../pattern-fonct/ex12-objetMetierView.js"></script>
19 --
20 <!-- Inclusion de jQuery pour les événements et manipulation du DOM -->
21 <script src="jquery.js"></script>
22 <!-- Mediator spécialisé pour filtrer les inputs (evt "change") -->
23 <script src="ex02-mediatorInputFilter.js"></script>
24 <!-- Génération automatique de formulaires avec filtrage des attributs -->
25 <script src="ex03-formsGui.js"></script>
26 --
27 <!-- Ajout d'une fonction de test, d'une méthode "main", et exécution -->
28 <script src="ex03-formsGuiTest.js"></script>

```

# Chapitre 5

## Exemple d'Application avec *IHM*

### 5.1 Principe de l'application et analyse fonctionnelle

Notre application, qui possède un *modèle* constitué d'une collection de personnes, permet (voir les *storyboards* sur la figure 5.1) :

- D'afficher la liste des noms de personnes (*items*) ;
- De sélectionner une personne en cliquant sur l'*item* correspondant (l'*item* est alors surligné et les détails concernant cette personne sont affichés) ;
- De modifier les données de la personnes (en l'occurrence le nom) en cliquant sur un bouton "*Modifier*".
- D'ajouter une personne ;
- De supprimer la personne sélectionnée.
- d'ajouter, de supprimer ou de modifier une adresse pour la personne sélectionnée.

Comme on peut le voir, nous avons une *agrégation* entre les personnes et les adresses, une personne pouvant avoir plusieurs adresses.

En recensant les événements (*clics* de boutons d'*items*, liens) possibles sur les *storyboards* de la figure 5.1, on dresse le diagramme de cas d'utilisation représenté sur la figure 5.2.

### 5.2 Modèle de donnée

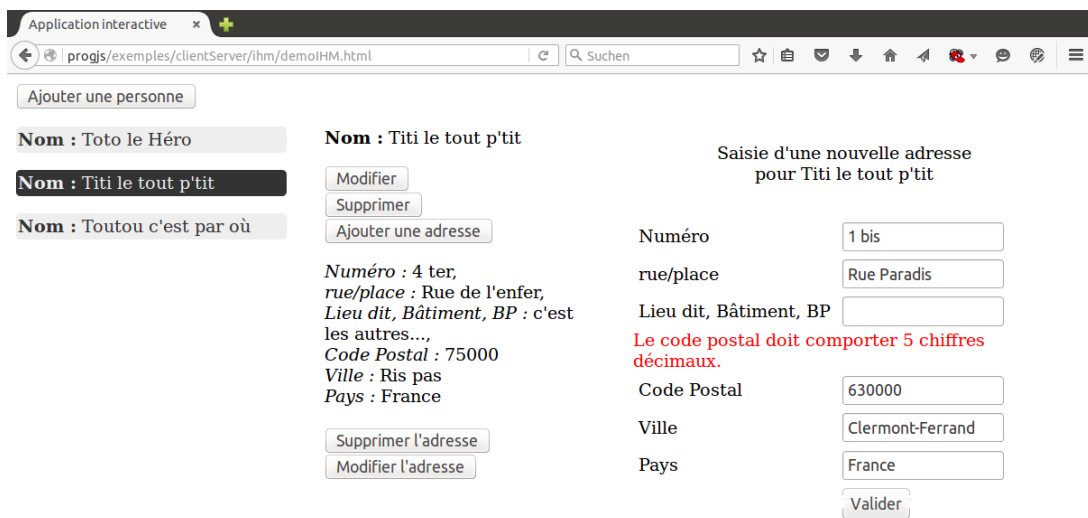
Dans notre modèle de données, une classe personne comporte un nom et une composition avec des instances d'adresse. Nous créons, pour le moment, quelques instances "en dur", dans un tableau **personnes**, avec chacune une adresse. Une autre propriété **selectedPersonne** contient une référence vers l'instance de personne sélectionnée (*item* surligné et détails affichés).

Code Source 5.1 : /ihm-demo/modelModule.js

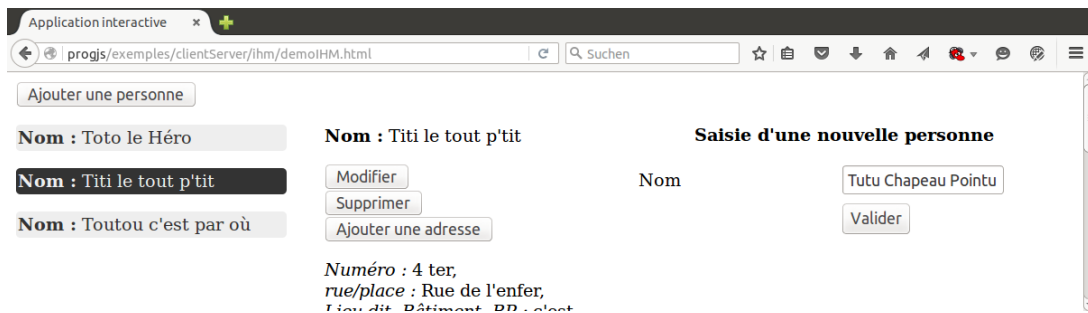
```
1 myApp.addModule.apply(myApp, [ "modele", {  
2   selectedPersonne : null ,  
3   personnes : [] ,  
4
```



(a) Sélection d'une personne (*item* surligné à gauche)



(b) Ajout d'une adresse pour la personne sélectionnée

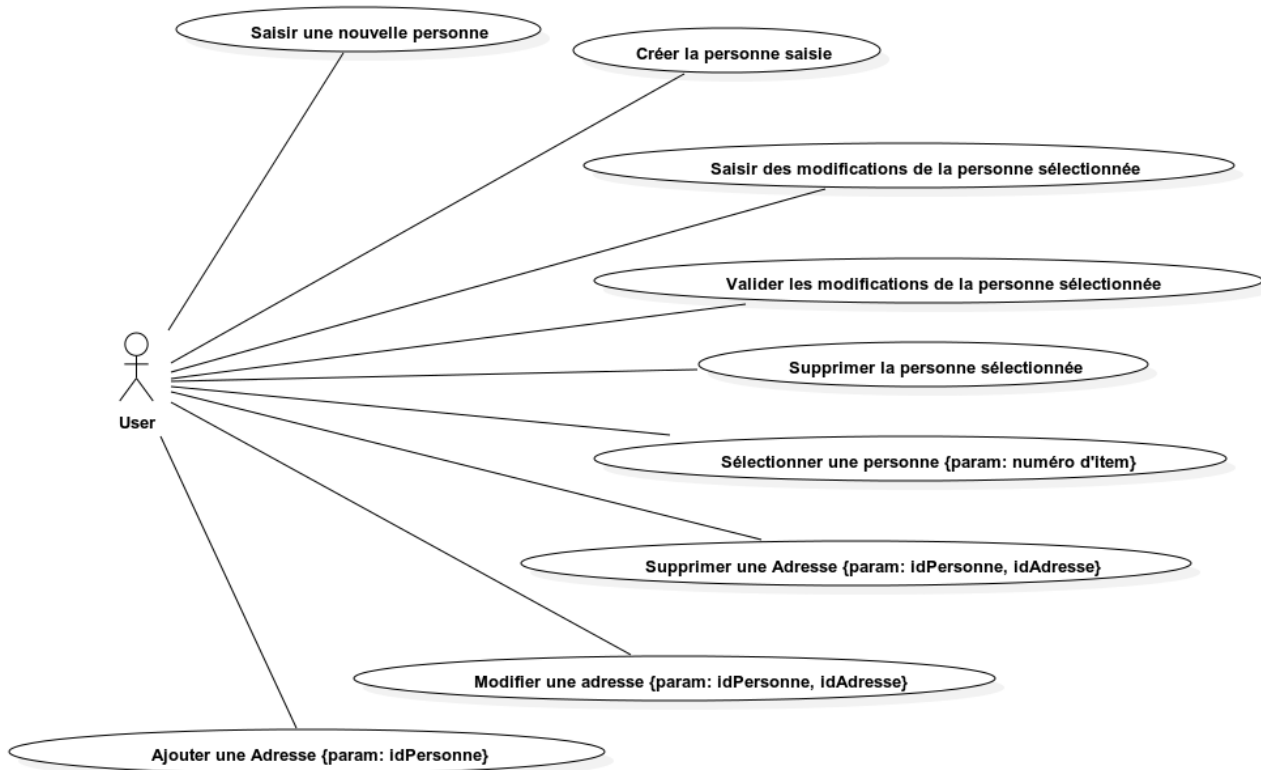


(c) Ajout d'une personne



(d) Après ajout d'une personne

FIGURE 5.1 : Captures d'écran de notre application

FIGURE 5.2 : Diagramme de cas d'utilisation de notre application avec son *IHM*

```

5   });
6
7   myApp.modele.personnes.push(myApp.metier.personne.createInstance({
8     id: "0123abcdef",
9     nom: "Toto le Héro",
10    adresse: myApp.metier.adresse.createInstance({
11      id: "04abf85bc9",
12      numeroRue: "2 bis",
13      rue: "Rue de l'a Paix",
14      complementAddr: "Dalle",
15      codePostal: "63000",
16      ville: "Clermont-Ferrand",
17      pays: "France 2"
18    })
19  }));
20
21  myApp.modele.personnes.push(myApp.metier.personne.createInstance({
22    id: "0123abcd12",
23    nom: "Titi le tout p'tit",
24    adresse: myApp.metier.adresse.createInstance({
25      id: "04abf85bb5",
26      numeroRue: "4 ter",
27      rue: "Rue de l'enfer",
28      complementAddr: "c'est les autres...",
29      codePostal: "75000",
30      ville: "Ris pas",
31      pays: "France"
32    })
  })

```

```

33     }));
34
35 myApp.modele.personnes.push(myApp.metier.personne.createInstance({
36     id : "0123abcd01",
37     nom : "Toutou c'est par où",
38     adresse : myApp.metier.adresse.createInstance({
39         id : "04abf85ba4",
40         numeroRue : "1",
41         rue : "Place de l'Alternative",
42         complementAddr : "Pourquoi pas",
43         codePostal : "63123",
44         ville : "Les Paumiers",
45         pays : "France"
46     })
47 }));

```

### 5.3 *Pattern Mediator* : centraliser les événements

Notre module `mediator` va nous permettre :

- De découpler l'implémentation de la réaction aux événements utilisateurs (modification du modèle, mise à jour des vues) de la gestion de ces événements utilisateurs via la technologie *jQuery*, qui, de ce fait, se trouve circonscrite à une seule classe (*Wrapper*).
- D'éliminer les dépendances cycliques entre les modules de notre application ;
- De recenser les événements utilisateurs de manière lisible dans un module centralisé ;
- De provoquer des mises à jour de panneaux de la vue qui observent des propriétés du modèle.

Contrairement au médiateur spécialisé dans le filtrage des attributs de formulaires décrit dans la partie 4.2, le module `mediator` va nous permettre d'exécuter plusieurs *callback* en réaction à un même événement (par exemple pour mettre à jour différentes parties de la vue après une modification du modèle).

Code Source 5.2 : `/ihm-demo/mediator.js`

```

1  /**
2   * Implémentation du pattern "Médiateur" pour la gestion des événements
      utilisateurs ,
3   * et la mise à jour des vues (ou des sous-arbres du DOM)
4   */
5  myApp.addModule.apply(myApp.gui, [ "mediator", function() {
6
7      /**
8       * Liste des événements pour lesquels une liste de callbacks peut être
          enregistrée
9       * @private
10      */
11      var m_subscriptionLists ;
12
13      /**

```



```
14  * Initialise la liste des événements, avec pour chacun, une liste de
15  * callbacks vide.
16  */
17  var init = function() {
18      m_subscriptionLists = {
19
20          // Opérations CRUD sur les personnes
21          "personne/read": [], // Lire toutes les personnes pour (re)onstruire le
                modèle
22
23          "personne/update": [], // validation du formulaire de mise à jour de la
                personne sélectionnée.
24          "personne/create": [], // validation du formulaire d'ajout d'une personne.
25          "personne/delete" : [], // Suppression d'une personne
26
27          // Opérations CRUD sur les adresses
28          "adresse/create": [], // validation du formulaire d'ajout d'une adresse.
29          "adresse/update" : [], // mise à jour d'une adresse
30          "adresse/delete" : [], // Suppression d'une adresse
31
32          // Actions Utilisateur donnant lieu à un changement de le vue
33          "personne/selectDetails" : [], // Sélection d'une personne pour voir les d
                étails
34          "personne/edit": [], // click sur la modification de la personne sé
                lectionnée
35          "personne/saisie": [], // click sur la modification de la personne sé
                lectionnée
36
37          "adresse/edit" : [], // Suppression d'une adresse
38          "adresse/saisie": [], // click sur la modification de la personne sé
                lectionnée
39
40          // Notifications de modification du modèle pour requête AJAX et/ou mise à
                jour de la vue
41          "personne/changed" : [], // mise à jour d'une personne
42          "personne/created" : [], // mise à jour d'une personne
43          "personne/detailsChanged": [], // Mise à jour requise du panneau des dé
                tails
44
45          // Notifications de modification du modèle pour requête AJAX et/ou mise à
                jour de la vue
46          "adresse/changed" : [], // mise à jour d'une adresse
47          "adresse/created" : [], // mise à jour d'une adresse
48
49
50          // Demande de ré-enregistrement d'événements utilisateurs suite à
                reconstruction d'éléments HTML
51          "personne/htmlListItemRebuilt": [], // Réenregistrement des événements de
                click sur les items
52
                // suite à reconstruction complète du
                code HTML des items.
53          "personne/detailsRebuilt": [], // Réenregistrement des événements de click
                sur les boutons "Supprimer", "Modifier"
54
                // suite à reconstruction du code HTML
                des détails.
```

```

55     };
56   };
57 };
58
59 // Appel de la méthode d'initialisation
60 init();
61
62 /**
63  * Interface publique du module mediator
64  */
65 var publicInterfaceMediator = {
66   /**
67    * Enregistrement d'un callback sur un événement.
68    * Il peut y avoir plusieurs callbacks sur un même événement
69    * (par exemple : mise à jour de deux parties distinctes de la vue)
70    * @param {string} eventCateg événement, qui doit être un nom de propriété
      de m_subscriptionLists
71    * @param {function} callbackFunction la fonction qui sera appelée en ré
      action à l'événement.
72    */
73   subscribe : function(eventCateg, callbackFunction){
74     if (m_subscriptionLists.hasOwnProperty(eventCateg)){
75       m_subscriptionLists[eventCateg].push({callback : callbackFunction});
76     }else{
77       throw new Error("Catégorie d'événements " + eventCateg + " inconnue du m
         édiateur");
78     }
79   },
80
81   /**
82    * Publication d'un événement survenu et exécution de tous les callbacks
      correspondants.
83    * @param {string} eventCateg événement, qui doit être un nom de propriété
      de m_subscriptionLists
84    * @param {Object} contextArg argument optionnel à transmettre au callback (
      exemple : item cliqué...)
85    */
86   publish : function(eventCateg, contextArg){
87     var i;
88     if (m_subscriptionLists.hasOwnProperty(eventCateg)){
89       for (var i=0 ; i< m_subscriptionLists[eventCateg].length ; ++i){
90         // On appelle le callback avec son
91         m_subscriptionLists[eventCateg][i].callback(contextArg);
92       }
93     }else{
94       throw new Error("Catégorie d'événements " + eventCateg + " inconnue du m
         édiateur");
95     }
96   },
97
98   // Réinitialise les listes de callbacks à vide.
99   empty : function(){
100     init();
101   }
102 };
103

```

```

104   return publicInterfaceMediator ;
105 }() );

```

## 5.4 Événements concernant les personnes

### 5.4.1 Enregistrement des événements utilisateurs via *jQuery*

Tous les événements recensés dans le **diagramme de cas d'utilisation** (voir la figure 5.2) se verront ici attribué un gestionnaire qui, généralement, ne fera que publier l'événement auprès de *mediator* (partie 5.3). Les éléments *HTML* constants de la vue (`<span>`, `<button>`, `<div>`, `<p>`, etc.) sur lesquels ces événements seront appliqués sont définis dans le fichier *HTML* principal décrit dans la partie 5.4.9.

Ces événements utilisateurs doivent parfois être réenregistrés suite à la reconstruction des éléments *HTML* concernés. Les événements sont alors détruits (méthodes `jQuery.off()`, ou `jQuery.empty()`, ou encore `jQuery.remove()`), puis, le code *HTML* est régénéré, et enfin, les événements utilisateur sont ré-enregistrés (méthode `jQuery.on()`).

S'il faut prévoir de ré-enregistrer un gestionnaire d'événement utilisateur, nous allons permettre de déclencher ce ré-enregistrement via le *mediator*. Ceci permet d'éviter notamment des problèmes de dépendance cyclique des fonctions *JavaScript* ou modules, par exemples du fait que les événements *jQuery* doivent être initialisés après la génération de la vue.

Code Source 5.3 : `/ihm-demo/guijQueryEventsPersonne.js`

```

1  /**
2   * Méthode d'initialisation des événements utilisateurs JavaScript.
3   * Enregistrement des gestionnaires de ces événements via jQuery.
4   */
5  myApp.addModule.apply(myApp.gui, [ "initjQueryEventsPersonne", function() {
6
7      //////////////////////////////////////
8      // click sur le bouton "Ajouter une personne" faisant sortir le formulaire
9
10     /**
11      * Gestionnaire click sur le bouton faisant sortir le formulaire
12      */
13     var clickBoutonSaisiePersonne = function(event){
14         // publication auprès du médiateur
15         myApp.gui.mediator.publish( "personne/saisie", {
16             personne : myApp.modele.selectedPersonne
17         });
18     };
19
20     // Enregistrement du Handler du click pour modifier les détails de l'item sélectionné via jQuery
21     $("#boutonAjouterPersonne").on( "click", clickBoutonSaisiePersonne );
22
23     //////////////////////////////////////
24     // click sur le bouton "Modifier le nom" faisant sortir le formulaire
25
26     /**
27      * Gestionnaire click sur le bouton faisant sortir le formulaire
28      */

```

```

29  var clickBoutonModifierPersonne = function(event){
30
31      // publication auprès du médiateur
32      myApp.gui.mediator.publish("personne/edit", {
33          personne : myApp.modele.selectedPersonne
34      });
35  };
36
37  //////////////////////////////////////
38  // click sur le bouton "Supprimer la personne" faisant sortir le formulaire
39
40  /**
41   * Gestionnaire click sur le bouton faisant sortir le formulaire
42   */
43  var clickBoutonSupprimerPersonne = function(event){
44      // publication auprès du médiateur
45      myApp.gui.mediator.publish("personne/delete", {
46          personne : myApp.modele.selectedPersonne
47      });
48  };
49
50
51  //////////////////////////////////////
52  // Gestionnaire de submit formulaire de modification de personne.
53
54  /**
55   * Gestionnaire de l'événement submit du formulaire.
56   * @param {Event} jQuery event correspondant au handler.
57   */
58  var formHandlerModifPersonne = function(event){
59
60      // Éviter d'appeler l'"action" par défaut () script PHP, etc...)
61      // du formulaire lors du submit
62      event.preventDefault();
63
64      // publication auprès du médiateur
65      myApp.gui.mediator.publish("personne/update", {
66          personne : myApp.modele.selectedPersonne
67      });
68  } // fin du gestionnaire formHandlerModif()
69
70  // Enregistrement du Handler du submit du formulaire via jQuery
71  $("#modifierPersonneForm").on("submit", formHandlerModifPersonne);
72
73
74  //////////////////////////////////////
75  // Gestionnaire de submit formulaire d'ajout de personne.
76
77  /**
78   * Gestionnaire de l'événement submit du formulaire.
79   * @param {Event} jQuery event correspondant au handler.
80   */
81  var formHandlerAjoutPersonne = function(event){
82
83      // Éviter d'appeler l'"action" par défaut () script PHP, etc...)
84      // du formulaire lors du submit

```

```

85     event.preventDefault();
86
87     // publication auprès du médiateur
88     myApp.gui.mediator.publish("personne/create", {
89         personne : myApp.modele.selectedPersonne
90     });
91 } // fin du gestionnaire formHandlerAjout()
92
93 // Enregistrement du Handler du submit du formulaire via jQuery
94 $("#ajouterPersonneForm").on("submit", formHandlerAjoutPersonne);
95
96
97 /**
98  * Enregistre les événements de clicks sur les boutons "Modifier" et "
99  * Supprimer"
100  * la personne sélectionnée.
101  * Cette fonction doit être invoquée en cas de sélection d'une nouvelle
102  * personne
103  * (reconstruction du code HTML du panneau des détails.
104  */
105 var registerButtonClickEvents = function(){
106     // Enregistrement du Handler du click pour modifier les détails de l'item sé
107     // lectionné via jQuery
108     $("#boutonModifierPersonne").on("click", clickBoutonModifierPersonne);
109     // Enregistrement du Handler du click pour supprimer l'item sélectionné via
110     // jQuery
111     $("#boutonSupprimerPersonne").on("click", clickBoutonSupprimerPersonne);
112 }
113
114 ///////////////////////////////////////////////////
115 // Clicks sur les éléments de la liste d'items
116
117 /** Méthode qui permet de créer un gestionnaire d'événement de click
118  * sur chaque nom de personnes (sélection des détails)
119  * Ces gestionnaires publient l'événement "nouvelle personne sélectionnée"
120  * auprès du médiateur.
121  * @param {int} index indice de l'item pour lequel on enregistre l'événement.
122  */
123 var registerHelperSelectDetails = function(index){
124     return function(){
125         myApp.gui.mediator.publish("personne/selectDetails",
126             {
127                 personne : myApp.modele.personnes[index]
128             });
129     };
130 };
131
132 /**
133  * Enregistre les événements javascript de click sur les éléments de la liste
134  * (noms des personnes).
135  * Cette méthode doit être appelée lors de la régénération du code de la liste
136  * .
137  * @method registerListePersonnesClicks
138  * @param {Object} contextArgs non utilisé
139  * @return {function} une fonction callback qui gère le click sur l'item
140  * index

```

```

134  */
135  var registerListePersonnesClicks = function(contextArgs){
136      for (var i=0 ; i<myApp.modele.personnes.length ; ++i){
137          $( "#master_" + myApp.modele.personnes[i].getId() ).on(
138              "click", registerHelperSelectDetails(i));
139      }
140  };
141
142  // Enregistrer les clicks lors de l'initialisation
143  registerButtonClickEvents();
144  registerListePersonnesClicks();
145
146  // Permet à la méthode qui régénère toute la liste des items
147  // de recréer, via le médiateur, les événements "click" sur les items.
148  myApp.gui.mediator.subscribe("personne/htmlListeItemRebuilt",
149      registerListePersonnesClicks);
150
151  // Permet à la méthode qui régénère le panneau des détails de recréer,
152  // via le médiateur, les événements "click" sur les boutons dans le panneau des
153  // détails.
154  myApp.gui.mediator.subscribe("personne/detailsRebuilt",
155      registerButtonClickEvents);
156  });

```

### 5.4.2 Mise à jour du panneau des détails

Le panneau des détails de l'*item* sélectionné doit être mis à jour lors de la modification de la personne par validation du formulaire, ou lors du changement de l'*item* sélectionné (*click* sur un autre *item*). dans ce cas, les événements utilisateurs sur les éléments *HTML* qui sont générés dynamiquement sur le panneau des détails doivent aussi être reconstruit (événement *personne/detailsRebuilt* du *mediator*).

Code Source 5.4 : /ihm-demo/guiDetailsChanged.js

```

1  /**
2   * Définition et enregistrement des callbacks de mise à jour des détails de l'
3   * item sélectionnée.
4   */
5  myApp.addModule.apply(myApp.gui, [ "callbacksUpdateDetails", function() {
6
7      /**
8       * Génération du code HTML des détails de la personne sélectionnée.
9       */
10     var getHtmlCodeDetail = function() {
11         var htmlCode = "<span class='panel'>" +
12             "<p><strong>Nom </strong>" + myApp.modele.selectedPersonne.getNom()
13             + "</p>" +
14             "<button id='boutonModifierPersonne'>Modifier</button><br/>" +
15             "<button id='boutonSupprimerPersonne'>Supprimer</button><br/>" +
16             "<button id='boutonAjouterAdresse'>Ajouter une adresse</button>";
17         for (var index = 0 ; index < myApp.modele.selectedPersonne.getNbAdresses() ;
18             ++index) {
19             htmlCode += "<p>" +
20                 myApp.view.adresse.getHtmlDevelopped(myApp.modele.
21                     selectedPersonne.getAdresse(index))

```

```

18         + "<br/><button id=\"boutonSupprimerAdresse_\"
19         + myApp.modele.selectedPersonne.getAdresse(index).getAttribute
            ( 'id ' )
20         + "\">Supprimer l'adresse</button>\"
21         + "<br/><button id=\"boutonModifierAdresse_\"
22         + myApp.modele.selectedPersonne.getAdresse(index).getAttribute
            ( 'id ' )
23         + "\">Modifier l'adresse</button>\"
24         + "</p>\";
25     }
26     htmlCode += "</span>\";
27     return htmlCode;
28 };
29
30 /**
31  * Redessine les détails d'une personne suite à sa sélection ou sa
      modification.
32  * @param {Object} contextArg non utilisé.
33  */
34 var repaintDetail = function(contextArg){
35
36     $("#modifierPersonneForm").empty(); // Vider les inputs et les événements JS
      existant
37     $("#ajouterPersonneForm").empty(); // Vider les inputs et les événements JS
      existant
38     $("#ajouterAdresseForm").empty(); // Vider les inputs et les événements JS
      existant
39     $("#modifierAdresseForm").empty(); // Vider les inputs et les événements JS
      existant
40
41     $("#vueDetail").empty(); // Vider les détails de l'item sélectionné
42
43     $("#vueDetail").html(getHtmlCodeDetail()); // Génération du code HTML
44
45     // Recréer les événements de clicks sur les boutons "modifier", "supprimer",
      etc.
46     myApp.gui.mediator.publish("personne/detailsRebuilt");
47 };
48
49 // Enregistrement du callback de l'événement dédié (m.a.j. des détails)
50 myApp.gui.mediator.subscribe("personne/detailsChanged", repaintDetail);
51 // Enregistrement du callback de l'événement de mise à jour de la personne
52 myApp.gui.mediator.subscribe("personne/changed", repaintDetail);
53
54 }());

```

### 5.4.3 Mise à jour du panneau des *items*

Le panneau qui affiche la liste des *items* doit être mis à jour lors de la modification de la personne par validation du formulaire (le nom de la personne peut changer), ou lors du changement de l'*item* sélectionné, celui-ci étant surligné.

En cas de changement de l'*item* sélectionné, la propriété `selectedPersonne` du modèle sera modifiée, et le rafraîchissement du panneau des détails sera ensuite provoqué.

Lors de la création d'une nouvelle personne, celle-ci sera automatiquement sélectionnée.

Code Source 5.5 : /ihm-demo/guiPersonneChanged.js

```

1  /**
2   * Définition et abonnement des callbacks de mise à jour de la
3   * liste clickable des items, soit lors de la modification
4   * du modèle, soit lors du changement de personne sélectionnée.
5   */
6  myApp.addModule.apply(myApp.gui, [ "callbacksMainListUpdate", function() {
7
8   /**
9    * Active ou désactive le surlignage (style CSS) d'un item de la liste.
10   * @param {personne} personne item de la liste à modifier (via l'ID de l'élément HTML).
11   * @param {boolean} highlighted true si on doit surligner, false pour remettre le style par défaut.
12   */
13   var setHighlighted = function(personne, highlighted){
14     if (highlighted){
15       // Mettre le style surligné sur l'item de la liste
16       $("#master_" + personne.getId()).css("background-color", "#333")
17         .css("color", "#eee")
18         .css("border-radius", "4px")
19         .css("padding", "2px");
20     } else {
21       // Remettre le style normal sur l'item de la liste
22       $("#master_" + personne.getId()).css("background-color", "#eee")
23         .css("color", "#333")
24         .css("border-radius", "4px")
25         .css("padding", "2px");
26     }
27   }
28
29   /**
30    * Génération du code HTML de la liste de personnes
31    */
32   var getHtmlCodeListePersonnes = function(){
33     var htmlCode = "";
34     for (var i=0 ; i<myApp.modele.personnes.length ; ++i){
35       htmlCode +=
36         "<p id=\"master_\"+ myApp.modele.personnes[i].getId() + \">\" +
37         "<strong>Nom </strong> \" + myApp.modele.personnes[i].getNom() + "</p>";
38     }
39     return htmlCode;
40   };
41
42   /**
43    * Raffraichissement (ou affichage) de toute la vue.
44    * @param contextArg non utilisé.
45    */
46   var repaintVue = function(contextArg){
47
48     $("#listePersonnes").empty(); // Vider la liste et ses événements
49     $("#listePersonnes").html(getHtmlCodeListePersonnes()); // afficher
50
51     // Appliquer le style par défaut sur tous les items
52     for (var i=0 ; i < myApp.modele.personnes.length ; ++i){
53       setHighlighted(myApp.modele.personnes[i], false);

```



```
54     }
55     // Surligner l'item s lectionn 
56     setHighlighted(myApp.modele.selectedPersonne, true);
57
58     // Recr er les  v nements de clicks sur les items de la liste
59     myApp.gui.mediator.publish("personne/htmlListItemRebuilt");
60 };
61
62 /**
63  * Changer l'item s lectionn  en r action   un click.
64  * @param {Object} contextArg argument indiquant la nouvelle personne s 
        lectionn e.
65  * @param {personne} contextArg.personne nouvelle personne s lectionn e.
66  */
67 var selectPersonne = function(contextArg){
68     // Supprimer le surlignage de l'ancienne personne s lectionn e
69     setHighlighted(myApp.modele.selectedPersonne, false);
70
71     // Changer l'item s lectionn 
72     myApp.modele.selectedPersonne = contextArg.personne;
73
74     // Mettre le style surlign  sur l'item s lectionn  de la liste
75     setHighlighted(myApp.modele.selectedPersonne, true);
76
77     // Provoquer la mise   jour du panneau des d tails
78     myApp.gui.mediator.publish("personne/detailsChanged", {
79         personne : myApp.modele.selectedPersonne
80     });
81 };
82
83 /**
84  * Changer l'item s lectionn  suite   cr ation d'une personne et mise   jour
        de la vue.
85  * @param {Object} contextArg argument indiquant la nouvelle personne s 
        lectionn e.
86  * @param {personne} contextArg.personne nouvelle personne s lectionn e.
87  */
88 var selectPersonneAnRepaint = function(contextArg){
89     selectPersonne(contextArg);
90     repaintVue();
91 }
92
93 // Enregistrement du callback de modification de la personne
94 myApp.gui.mediator.subscribe("personne/changed", repaintVue);
95 // Enregistrement du callback de cr ation de la personne
96 myApp.gui.mediator.subscribe("personne/created", selectPersonneAnRepaint);
97
98 // Enregistrement du callback de s lection d'une nouvelle personne.
99 myApp.gui.mediator.subscribe("personne/selectDetails", selectPersonne);
100 }());
```

#### 5.4.4 Bouton "Supprimer"

Lorsque l'utilisateur clique sur "Supprimer", la personne s lectionn e est supprim e du mod le. Une nouvelle personne est s lectionn e (personne par d faut) et la vue est r initialis e.

Code Source 5.6 : /ihm-demo/guiBoutonSupprimerPersonne.js

```

1  /**
2   * Définition et enregistrement des callbacks appelés à gérer le clic sur le
      bouton
3   * "modifier" la personne sélectionnée.
4   */
5  myApp.addModule.apply(myApp.gui, [ "callbacksClickSupprimer", function() {
6
7      /**
8       * Callback qui supprime la personne passée dans l'objet passé en argument.
9       * @param {Object} contextArg argument indiquant la personne à supprimer.
10      * @param {personne} contextArg.personne référence de l'instance de personne à
      supprimer dans le modèle.
11      */
12      var deletePersonne = function(contextArg){
13          // Indice dans le tableau de la personne à supprimer.
14          var indexSelectedPersonne = myApp.modele.personnes.indexOf(contextArg.
      personne);
15          // Suppression de la personne dans le modèle
16          myApp.modele.personnes.splice(indexSelectedPersonne, 1);
17          // Personne sélectionnée par défaut
18          myApp.modele.selectedPersonne = myApp.modele.personnes[0];
19
20          // Provoquer la mise à jour de la vue :
21          myApp.gui.mediator.publish( "personne/changed", {
22              personne : myApp.modele.selectedPersonne
23          });
24      }
25
26      // Enregistrement du callback
27      myApp.gui.mediator.subscribe( "personne/delete", deletePersonne );
28
29  }()) );

```

#### 5.4.5 Bouton "Modifier" et affichage du formulaire

Lorsque l'utilisateur clique sur "Modifier", le formulaire doit être affiché avec les données de la personnes dans les *inputs*.

Code Source 5.7 : /ihm-demo/guiBoutonModifierPersonne.js

```

1  /**
2   * Définition et enregistrement des callbacks appelés à gérer le clic sur le
      bouton
3   * "modifier" la personne sélectionnée.
4   */
5  myApp.addModule.apply(myApp.gui, [ "callbacksClickModifierPersonne", function() {
6
7      /**
8       * Génération du code HTML du formulaire de modification de la personne sé
      lectionnée.
9       */
10     var getHtmlFormInputs = function() {
11         return "<span style=\"width :360px; display : inline-block; vertical-align :
      top ;\>" +

```

```

12         myApp.gui.getHtmlFormInputs(myApp.modele.selectedPersonne, "
13             modifierPersonneForm") +
14         "<label></label><input type='submit' value='Valider'></input>" +
15         "</span>";
16     }
17     /**
18      * Callback d’Affichage (via le DOM) du formulaire dans l’ l ment d’ID "
19      * modifierPersonneForm"
20      * @param {Object} contextArg non utilis .
21      */
22     var repaintFormInputs = function(contextArg){
23         $("#modifierPersonneForm").empty(); // Vider les inputs et les  v nements JS
24         // existant
25         $("#ajouterPersonneForm").empty(); // Vider les inputs et les  v nements JS
26         // existant
27         $("#ajouterAdresseForm").empty(); // Vider les inputs et les  v nements JS
28         // existant
29         $("#modifierAdresseForm").empty(); // Vider les inputs et les  v nements JS
30         // existant
31         $("#modifierPersonneForm").append(getHtmlFormInputs()); // ajouter les
32         // nouveaux inputs
33     };
34
35     // Enregistrement du callback
36     myApp.gui.mediator.subscribe("personne/edit", repaintFormInputs);
37 }

```

### 5.4.6 Bouton "Ajouter une personne"

Lorsque l’utilisateur clique sur "Ajouter une personne", le formulaire doit  tre affich  avec les valeurs par d faut (typiquement des champs vides) dans les *inputs*.

Pour cela, on utilise la possibilit  offerte par la fabrique de nos modules m tier (partie 2.6.3) de cr er un objet par d faut en passant `null` en argument de la fabrique. Ceci permet de ne pas g n rer de messages d’erreur en cas de champs obligatoire initialement vide.

Apr s validation du formulaire, la personne est ajout e dans le mod le, elle est automatiquement s lectionn e, et la vue est mise   jour.

Code Source 5.8 : `/ihm-demo/guiBoutonAjouterPersonne.js`

```

1  /**
2   * D finition et enregistrement des callbacks appel s   g rer le clic sur le
3   * bouton
4   * "modifier" la personne s lectionn e.
5   */
6  myApp.addModule.apply(myApp.gui, ["callbacksClickAjouter", function(){
7
8   /**
9   * G n ration du code HTML du formulaire de modification de la personne s 
10  * lectionn e.
11  */
12  var getHtmlFormInputs = function(){

```

```

11     return "<span style=\|width :360px; display : inline-block; vertical-align :
12         top ;\|>" +
13         "<strong style=\|width : 360px; display : inline-block; text-align :
14             center; padding : 15px;\|>Saisie d'une nouvelle personne</strong>"
15         +
16         myApp.gui.getHtmlFormInputs(myApp.metier.personne.createInstance(
17             null), "ajouterPersonneForm") +
18         "<label></label><input type=\|submit\| value=\|Valider\|</input>" +
19         "</span>";
20 }
21
22 /**
23  * Callback d’Affichage (via le DOM) du formulaire dans l’élément d’ID "
24     mainForm"
25  * @param {Object} contextArg non utilisé.
26  */
27 var repaintFormInputs = function(contextArg){
28     $("#modifierPersonneForm").empty(); // Vider les inputs et les événements JS
29     // existant
30     $("#ajouterPersonneForm").empty(); // Vider les inputs et les événements JS
31     // existant
32     $("#ajouterAdresseForm").empty(); // Vider les inputs et les événements JS
33     // existant
34     $("#modifierAdresseForm").empty(); // Vider les inputs et les événements JS
35     // existant
36     $("#ajouterPersonneForm").append(getHtmlFormInputs()); // ajouter les
37     // nouveaux inputs
38 };
39
40 // Enregistrement du callback
41 myApp.gui.mediator.subscribe("personne/saisie", repaintFormInputs);
42
43 }();

```

### 5.4.7 Validation du formulaire de modification

Lors de la validation (événement *submit*) du formulaire de modification, les données de la personne sélectionnée doivent être mises à jour à partir des valeurs saisies dans le formulaire. Les panneaux potentiellement impactés (liste des *items*, panneau des détails) sont alors mis à jour.

Code Source 5.9 : /ihm-demo/guiModifierPersonneFormValidate.js

```

1  /**
2   * Définition et enregistrement du callback réagissant à la validation (submit)
3   * du formulaire de modification d'une personne.
4   */
5  myApp.addModule.apply(myApp.gui, ["callbacksValidateModifierForm", function(){
6      // Formulaire de modification d'une personne
7
8      /**
9       * Modifie le modèle à partir des données saisies dans le formulaire
10      */
11      var updateModel = function(){

```

```

12
13 // 1) Mise à jour des données du modèle
14 // à partir des valeurs des inputs du formulaire
15 var attributeName,
16     inputId;
17 // On récupère les attributs du formulaire dans une nouvelle instance
18 var changedPersonne = myApp.metier.personne.createInstance(null);
19 // Pour chaque propriété (chaque input du formulaire)
20 for (var j=0 ; j< myApp.metier.personne.getAttributeList().length ; ++j){
21     var attributeName = myApp.metier.personne.getAttributeList()[j];
22     if (attributeName !== "id"){
23         // calcul de l'ID de l'input
24         var inputId = myApp.gui.getInputId({
25             attributeName : attributeName,
26             formId : "modifierPersonneForm"
27         });
28         // Modification de la propriété de la personne
29         // avec la valeur saisie dans l'input.
30         changedPersonne.setAttribute(attributeName,
31             document.getElementById(inputId).value
32         );
33     }
34 }
35 // Seulement s'il n'y a pas d'erreur (filtrage strict côté client)
36 if (!changedPersonne.hasError()){
37     for (var j=0 ; j< myApp.metier.personne.getAttributeList().length ; ++j){
38         var attributeName = myApp.metier.personne.getAttributeList()[j];
39         if (attributeName !== "id"){
40             myApp.modele.selectedPersonne.setAttribute(attributeName,
41                 changedPersonne.getAttribute(
42                     attributeName));
43         }
44     }
45     // Provoquer la mise à jour des éléments de la vue observant la personne
46     myApp.gui.mediator.publish("personne/changed", {
47         personne : myApp.modele.
48             selectedPersonne
49     });
50 };
51
52 // Enregistrement du callback de l'événement de validation du formulaire
53 myApp.gui.mediator.subscribe("personne/update", updateModel);
54 }());

```

### 5.4.8 Validation du formulaire d'ajout d'une personne

Lors de la validation (événement *submit*) du formulaire d'ajout, une personne doit être ajoutée au modèle à partir des valeurs saisies dans le formulaire. Les panneaux potentiellement impactés (liste des *items*, panneau des détails) sont alors mis à jour.

Code Source 5.10 : `/ihm-demo/guiAjouterPersonneFormValidate.js`

1 `/**`

```

2  * Définition et enregistrement du callback réagissant à la validation (submit)
3  * du formulaire de modification d'une personne.
4  */
5  myApp.addModule.apply(myApp.gui, [ "callbacksValidateAjouterForm", function() {
6
7      /**
8       * Modifie le modèle à partir des données saisies dans le formulaire
9       */
10     var updateModel = function() {
11         // 1) Mise à jour des données du modèle
12         // à partir des valeurs des inputs du formulaire
13         var attributeName,
14             inputId;
15
16         // Récupération des données du formulaire dans une nouvelle instance
17         var nouvellePersonne = myApp.metier.personne.createInstance(null);
18         // Pour chaque propriété (chaque input du formulaire)
19         for (var j=0 ; j< myApp.metier.personne.getAttributeList().length ; ++j){
20             var attributeName = myApp.metier.personne.getAttributeList()[j];
21             if (attributeName != "id"){
22                 // calcul de l'ID de l'input
23                 var inputId = myApp.gui.getInputId({
24                     attributeName : attributeName,
25                     formId : "ajouterPersonneForm"
26                 });
27                 // Modification de la propriété de la personne
28                 // avec la valeur saisie dans l'input.
29                 nouvellePersonne.setAttribute(attributeName,
30                     document.getElementById(inputId).value
31                 );
32             }
33         }
34     }
35     // Seulement s'il n'y a pas d'erreur (filtrage strict côté client)
36     if (!nouvellePersonne.hasError()) {
37         // Ajout de la Personne au modèle
38         myApp.modele.personnes.push(nouvellePersonne);
39         // Provoquer la sélection de la nouvelle personne (et par suite la mise à
40         // jour de la vue)
41         myApp.gui.mediator.publish("personne/created", {
42             personne : nouvellePersonne
43         });
44     }
45 }
46 // Enregistrement du callback de l'événement de validation du formulaire
47 myApp.gui.mediator.subscribe("personne/create", updateModel);
48
49 }());

```

#### 5.4.9 Code *HTML* de la vue et invocation des méthodes

Il faut surtout penser à inclure `jqueryjs` le plus tard possible et à invoquer la méthode d'enregistrement des événements utilisateurs après la génération de la vue, qui crée les éléments *HTML* sur lesquels on applique ces événements.

Code Source 5.11 : /ihm-demo/index.html

```

1 <!doctype HTML>
2 <html lang="fr">
3 <head>
4   <meta charset="UTF-8" />
5   <title>Application interactive</title>
6   <link rel="stylesheet" href="basicStyle.css"/>
7 </head>
8 <body>
9   <!-- Paragraphe contenant le résultats du script : -->
10  <p id="paragrapheResultat"></p>
11  <!-- Structure de l'application vide avec deux méthodes -->
12  <script src="../pattern-fonct/ex04-structureApplication.js"></script>
13  <!-- Création de sous-module regexUtil de myApp.metier -->
14  <script src="../pattern-fonct/ex05-modulePatternRegex.js"></script>
15  <!-- Sous-module adresse de myApp.metier -->
16  <script src="../pattern-proto/ex05-createModuleMetierProto.js"></script>
17  <!-- Méthode fabrique générique d'objets métier -->
18  <script src="../pattern-proto/ex05-fabriqueObjetMetierProto.js"></script>
19  <!-- Création de sous-module adresse de myApp.metier -->
20  <script src="../pattern-fonct/ex06-moduleMetierAdresse.js"></script>
21  <!-- Création d'une méthode fabrique d'adresse de myApp.metier.adresse -->
22  <script src="../pattern-fonct/ex08-fabriqueAdresse.js"></script>
23  <!-- Classe de vérification de l'implémentation d'interfaces -->
24  <script src="../pattern-fonct/ex11-interfaceImplementation.js"></script>
25  <!-- Classe de vérification de l'implémentation d'interfaces -->
26  <script src="../pattern-fonct/ex11-interfaceImplementationMetier.js"></script>
27  <!-- Création de fonctions d'affichage dans myApp.metier.view.adresse -->
28  <script src="../pattern-fonct/ex09-adresseView.js"></script>
29
30  <!-- Inclusion de jQuery pour les événements et manipulation du DOM -->
31  <script src="jquery.js"></script>
32  <!-- Mediator spécialisé pour filtrer les inputs (evt "change") -->
33  <script src="../form-filter/ex02-mediatorInputFilter.js"></script>
34  <!-- Génération automatique de formulaires avec filtrage des attributs -->
35  <script src="../form-filter/ex03-formsGui.js"></script>
36
37  <!-- Module Métier myApp.metier.personnes avec collection getAdresses() -->
38  <script src="../personneModule.js"></script>
39  <!-- Construction en dur d'un modèle de données : collection de personnes -->
40  <script src="../modelModule.js"></script>
41  <!-- Pattern Médiateur pour enchainement architecture trois tiers -->
42  <script src="../mediator.js"></script>
43
44  <!-- Implémentation de l'action "click sur 'Modifier' de Personne" -->
45  <script src="../guiBoutonModifierPersonne.js"></script>
46  <!-- Implémentation de l'action "click sur 'Supprimer' de Personne" -->
47  <script src="../guiBoutonSupprimerPersonne.js"></script>
48  <!-- Implémentation de l'action "click sur 'Ajouter' de Personne" -->
49  <script src="../guiBoutonAjouterPersonne.js"></script>
50  <!-- Implémentation de l'action "click sur 'Ajouter' de Adresse" -->
51  <script src="../guiBoutonAjouterAdresse.js"></script>
52  <!-- Implémentation de l'action "click sur 'Ajouter' de Adresse" -->
53  <script src="../guiBoutonModifierAdresse.js"></script>
54  <!-- Implémentation de l'action "click sur 'Supprimer' de Adresse" -->
55  <script src="../guiBoutonSupprimerAdresse.js"></script>

```

```

56 <!-- Implémentaton de l'action "Validation de formulaire de modif" -->
57 <script src="/guiModifierPersonneFormValide.js"></script>
58 <!-- Implémentaton de l'action "Validation de formulaire d'ajout" -->
59 <script src="/guiAjouterPersonneFormValide.js"></script>
60 <!-- Implémentaton de l'action "Validation de formulaire de modif" -->
61 <script src="/guiModifierAdresseFormValide.js"></script>
62 <!-- Implémentaton de l'action "Validation de formulaire d'ajout" -->
63 <script src="/guiAjouterAdresseFormValide.js"></script>
64 <!-- Mise à jour de la vue (panneau "détails" uniquement) -->
65 <script src="/guiDetailsChanged.js"></script>
66 <!-- Mise à jour de la vue (re-générer toute la vue) -->
67 <script src="/guiPersonneChanged.js"></script>
68
69 <!-- Code HTML de la vue -- Structure générale de la page HML -->
70
71 <button id="boutonAjouterPersonne">Ajouter une personne</button><br/>
72 <span id="listePersonnes" class="panel"></span>
73 <span class="panel">
74   <span id="vueDetail">
75     </span><br/><br/>
76 </span>
77 <span id="spanMainForm" class="panel">
78   <form id="ajouterPersonneForm" method="post"></form>
79   <form id="modifierPersonneForm" method="post"></form>
80   <form id="ajouterAdresseForm" method="post"></form>
81   <form id="modifierAdresseForm" method="post"></form>
82 </span>
83
84 <!-- Inclusion de jQuery pour les événements et manipulation du DOM -->
85 <script src="/jquery.js"></script>
86 <!-- Événements utilisateurs concernant les personnes -->
87 <script src="/guiJQueryEventsPersonne.js"></script>
88 <!-- Événements utilisateurs concernant les adresses -->
89 <script src="/guiJQueryEventsAdresse.js"></script>
90
91 <!-- Ajout d'un main et exécution -->
92 <script>
93   /**
94    * Série d'instructions effectuées pour initialiser l'application/
95    * @method mainFunction
96    * @augments myApp
97    */
98   myApp.addModule("mainFunction", function() {
99
100     // Personne sélectionnée par défaut
101     myApp.modele.selectedPersonne = myApp.modele.personnes[0];
102
103     // Provoquer le premier affichage de la vue :
104     myApp.gui.mediator.publish("personne/changed", {
105       personne : myApp.modele.selectedPersonne
106     });
107
108     // Enregistrement des événements utilisateurs gérés par jQuery
109     myApp.gui.initJQueryEventsPersonne();
110     myApp.gui.initJQueryEventsAdresse();
111

```



```

112     });
113
114     //////////////////////////////////////
115     // Exécution du Main avec un test d'exception
116     // try{
117     //   Exécution de la méthode mainFunction
118     myApp.mainFunction();
119     // } catch (e){
120     //   alert(e.message);
121     // }
122 </script>
123 </body>
124 </html>

```

## 5.5 Événements concernant les Adresses

### 5.5.1 Enregistrement des événements utilisateurs via *jQuery*

De même que pour les personnes, l'utilisation de *jQuery* est limitée à un module *Wrapper*, qui va définir tous les *handler*.

Comme il peut y avoir plusieurs adresses, dont les éléments *HTML* sont générés dynamiquement, sur le panneau des détails, les événements concernant les adresses doivent pouvoir être reconstruits dans le cas d'une reconstruction du panneau des détails de la vue (événement *personne/detailsRebuilt* du *mediator*. De plus, nous devons créer un *handler* pour chacune des adresses de la personne sélectionnée. Ces *handler* seront créés grâce à des *helpers*.

Code Source 5.12 : */ihm-demo/guijQueryEventsAdresse.js*

```

1  /**
2   * Méthode d'initialisation des événements utilisateurs JavaScript.
3   * Enregistrement des gestionnaires de ces événements via jQuery.
4   */
5  myApp.addModule.apply(myApp.gui, [ "initjQueryEventsAdresse", function() {
6
7      /**
8       * Gestionnaire click sur le bouton faisant sortir le formulaire
9       */
10     var clickBoutonSaisieAdresse = function(event) {
11         // publication auprès du médiateur
12         myApp.gui.mediator.publish("adresse/saisie", {
13             personne : myApp.modele.selectedPersonne
14         });
15     };
16
17     /** Méthode qui permet de créer un gestionnaire d'événement de click
18      * du bouton de suppression sur chaque adresse de la personne sélectionnée.
19      * Ces gestionnaires publient l'événement "nouvelle personne sélectionnée"
20      * auprès du médiateur.
21      * @param {int} index indice de l'adresse pour lequel on enregistre l'évé-
22      * nement.
23      */
24     var registerHelperSupprimerAdresse = function(index) {
25         return function() {
26             myApp.gui.mediator.publish("adresse/delete",

```

```

25         {
26             personne : myApp.modele.selectedPersonne ,
27             adresse : myApp.modele.selectedPersonne.getAdresse(index)
28         });
29     };
30 };
31
32 /** Méthode qui permet de créer un gestionnaire d'événement de click
33 * du bouton de suppression sur chaque adresse de la personne sélectionnée.
34 * Ces gestionnaires publient l'événement "nouvelle personne sélectionnée"
35 * @param {int} index indice de l'adresse pour lequel on enregistre l'événement
36 */
37 var registerHelperModifierAdresse = function(index){
38     return function(){
39         myApp.gui.mediator.publish("adresse/edit",
40             {
41                 personne : myApp.modele.selectedPersonne ,
42                 adresse : myApp.modele.selectedPersonne.getAdresse(index)
43             });
44     };
45 };
46
47 /**
48 * Enregistre les événements de clicks sur les boutons "Ajouter une adresse"
49 * et
50 * les boutons "Supprimer" ou modifier de toutes les adresses de la personne s
51 * électionnée.
52 * Cette fonction doit être invoquée en cas de sélection d'une nouvelle
53 * personne
54 * (reconstruction deu code HTML du panneau des détails).
55 */
56 var registerButtonClickEvents = function(){
57     var idBoutonSupprimerAdresse ,
58         idBoutonModifierAdresse ;
59     // Enregistrement du Handler du click pour ajouter une adresse
60     $("#boutonAjouterAdresse").on("click", clickBoutonSaisieAdresse);
61
62     for (var i=0 ; i < myApp.modele.selectedPersonne.getNbAdresses() ; ++i){
63         idBoutonSupprimerAdresse = "boutonSupprimerAdresse_" +
64             myApp.modele.selectedPersonne.getAdresse(i).getAttribute('id');
65         $("#" + idBoutonSupprimerAdresse).on("click",
66             registerHelperSupprimerAdresse(i));
67         idBoutonModifierAdresse = "boutonModifierAdresse_" +
68             myApp.modele.selectedPersonne.getAdresse(i).getAttribute('id');
69         $("#" + idBoutonModifierAdresse).on("click", registerHelperModifierAdresse
70             (i));
71     }
72 }
73
74 //////////
75 // Gestionnaire de submit formulaire d'ajout de adresse.
76
77 /**
78 * Gestionnaire de l'événement submit du formulaire.

```

```
74  * @param {Event} jQuery event correspondant au handler.
75  */
76  var formHandlerAjoutAdresse = function(event){
77
78      //  viter d'appeler l'"action" par d faut () script PHP, etc...)
79      // du formulaire lors du submit
80      event.preventDefault();
81
82      // publication aupr s du m diator
83      myApp.gui.mediator.publish("adresse/create", {
84          personne : myApp.modele.selectedPersonne
85      });
86  } // fin du gestionnaire formHandlerAjout()
87
88  // Enregistrement du Handler du submit du formulaire via jQuery
89  $("#ajouterAdresseForm").on("submit", formHandlerAjoutAdresse);
90
91  //////////////////////////////////////
92  // Gestionnaire de submit formulaire d'ajout de adresse.
93
94  /**
95   * Gestionnaire de l' v nement submit du formulaire.
96   * @param {Event} jQuery event correspondant au handler.
97   */
98  var formHandlerModifierAdresse = function(event){
99
100     //  viter d'appeler l'"action" par d faut () script PHP, etc...)
101     // du formulaire lors du submit
102     event.preventDefault();
103
104     // publication aupr s du m diator
105     myApp.gui.mediator.publish("adresse/update", {
106         personne : myApp.modele.selectedPersonne
107     });
108  } // fin du gestionnaire formHandlerAjout()
109
110  // Enregistrement du Handler du submit du formulaire via jQuery
111  $("#modifierAdresseForm").on("submit", formHandlerModifierAdresse);
112
113  // Enregistrer les clicks lors de l'initialisation
114  registerButtonClickEvents();
115
116  // Permet   la m thode qui reg n re le panneau des d tails de recrer,
117  // via le m diator, les  v nements "click" sur les boutons dans le panneau des
118  // d tails.
119  myApp.gui.mediator.subscribe("personne/detailsRebuilt",
120      registerButtonClickEvents);
121  });
```

### 5.5.2 Boutons d'ajout, de suppression, et de modification

Le bouton d'ajout d'une adresse, qui existe un un seul exemplaire car il d pend uniquement de la personne, est le plus simple. Il faut cr er un formulaire vierge pour la saisie d'une adresse.

Comme pour une personne, on utilise la possibilit  de passer `null` comme argument de la

fabrique d'adresse, qui crée alors une adresse par défaut, sans créer d'erreurs pour les champs vides (même pour les champs obligatoires).

Code Source 5.13 : /ihm-demo/guiBoutonAjouterAdresse.js

```

1  /**
2   * Définition et enregistrement des callbacks appelés à gérer le clic sur le
      bouton
3   * "modifier" la personne sélectionnée.
4   */
5  myApp.addModule.apply(myApp.gui, [ "callbacksClickAjouter", function() {
6
7    /**
8     * Génération du code HTML du formulaire de modification de la personne sé-
      lectionnée.
9     */
10   var getHtmlFormInputs = function() {
11     return "<span style=\"width :360px; display : inline-block; vertical-align :
      top ;\">" +
12       "<p style=\"width : 360px; display : inline-block; text-align :center ;
      padding : 15px;\">" +
13       "<tr><td>Saisie d'une nouvelle adresse</td></tr>" +
14       "<br>pour " + myApp.modele.selectedPersonne.getAttribute("nom") + "
      </p>" +
15       myApp.gui.getHtmlFormInputs(myApp.metier.adresse.createInstance(null
      ), "ajouterAdresseForm") +
16       "<label></label><input type=\"submit\" value=\"Valider\"></input>" +
17       "</span>";
18   }
19
20   /**
21    * Callback d’Affichage (via le DOM) du formulaire dans l’élément d’ID "
      mainForm"
22    * @param {Object} contextArg non utilisé.
23    */
24   var repaintFormInputs = function(contextArg) {
25     $("#modifierPersonneForm").empty(); // Vider les inputs et les événements JS
      existant
26     $("#ajouterPersonneForm").empty(); // Vider les inputs et les événements JS
      existant
27     $("#ajouterAdresseForm").empty(); // Vider les inputs et les événements JS
      existant
28     $("#modifierAdresseForm").empty(); // Vider les inputs et les événements JS
      existant
29
30     $("#ajouterAdresseForm").append(getHtmlFormInputs()); // ajouter les
      nouveaux inputs
31   };
32
33   // Enregistrement du callback
34   myApp.gui.mediator.subscribe("adresse/saisie", repaintFormInputs);
35
36 }());

```

Les boutons de modification et de suppression des adresse doivent exister en autant d'exemple qu'il y a d'adresse. On crée donc un *helper* chargé de créer le *callback* correspondant à chaque adresse.

Code Source 5.14 : /ihm-demo/guiBoutonModifierAdresse.js

```

1  /**
2   * Définition et enregistrement des callbacks appelés à gérer le clic sur le
      bouton
3   * "modifier" la personne sélectionnée.
4   */
5  myApp.addModule.apply(myApp.gui, [ "callbacksClickModifierPersonne", function() {
6
7      /**
8       * Génération du code HTML du formulaire de modification de la personne sé-
          lectionnée.
9       */
10     var getHtmlFormInputs = function(adresse){
11         return "<span style=\"width :360px; display : inline-block; vertical-align :
            top ;\>" +
12             myApp.gui.getHtmlFormInputs(adresse, "modifierAdresseForm") +
13             "<label></label><input type=\"submit\" value=\"Valider\"></input>" +
14             "</span>";
15     }
16
17     /**
18      * Callback d’Affichage (via le DOM) du formulaire dans l’élément d’ID "
          modifierAdresseForm"
19      * Si l’adresse a des erreurs, potentiellement, elle n’a pas été créée sur le
          serveur.
20      * Il faut alors utiliser le verbe POST. On ouvre alors l’élément d’ID "
          ajouterAdresseForm"
21      * @param {Object} contextArg non utilisé.
22      */
23     var repaintFormInputs = function(contextArg){
24         $("#modifierPersonneForm").empty(); // Vider les inputs et les événements JS
          existant
25         $("#ajouterPersonneForm").empty(); // Vider les inputs et les événements JS
          existant
26         $("#ajouterAdresseForm").empty(); // Vider les inputs et les événements JS
          existant
27         $("#modifierAdresseForm").empty(); // Vider les inputs et les événements JS
          existant
28
29         $("#modifierAdresseForm").append(getHtmlFormInputs(
30             contextArg.adresse)); // ajouter les
          nouveaux inputs
31     };
32
33     // Enregistrement du callback
34     myApp.gui.mediator.subscribe("adresse/edit", repaintFormInputs);
35
36 }());

```

Code Source 5.15 : /ihm-demo/guiBoutonSupprimerAdresse.js

```

1  /**
2   * Définition et enregistrement des callbacks appelés à gérer le clic sur le
      bouton
3   * "modifier" la adresse sélectionnée.
4   */

```

```

5 myApp.addModule.apply(myApp.gui, [ "callbacksClickSupprimerAdresse", function() {
6
7     /**
8      * Callback qui supprime la adresse passée dans l'objet passé en argument.
9      * @param {Object} contextArg argument indiquant la adresse à supprimer.
10     * @param {adresse} contextArg.adresse référence de l'instance de adresse à
        supprimer dans le modèle.
11     */
12     var deleteAdresse = function(contextArg){
13
14         // Suppression de l'adresse dans la personne
15         contextArg.personne.deleteAdresse(contextArg.adresse);
16
17         // Provoquer la mise à jour de la vue :
18         myApp.gui.mediator.publish("personne/detailsChanged", {
19             adresse : myApp.modele.selectedPersonne
20         });
21     }
22
23     // Enregistrement du callback
24     myApp.gui.mediator.subscribe("adresse/delete", deleteAdresse);
25
26 }())];

```

### 5.5.3 Création d'une nouvelle adresse

L'adresse est automatiquement ajoutée à la personne sélectionnée, et son *ID* est généré automatiquement. Comme dans le cas d'une personne, les attributs de l'adresse (autre que l'*ID*) sont récupérées à partir des valeurs des *inputs* du formulaire.

Code Source 5.16 : /ihm-demo/guiAjouterAdresseFormValidate.js

```

1  /**
2   * Définition et enregistrement du callback réagissant à la validation (submit)
3   * du formulaire de modification d'une adresse.
4   */
5  myApp.addModule.apply(myApp.gui, [ "callbacksValidateAjouterAdresseForm",
        function(){
6      /**
7       * Modifie le modèle à partir des données saisies dans le formulaire
8       */
9      var updateModel = function(){
10         // 1) Mise à jour des données du modèle
11         // à partir des valeurs des inputs du formulaire
12         var attributeName,
13             inputId;
14
15         // Ajout d'une adresse vide dans la collection
16         var nouvelleAdresse = myApp.metier.adresse.createInstance(null);
17         myApp.modele.selectedPersonne.addAdresse(nouvelleAdresse);
18
19         // Pour chaque propriété (chaque input du formulaire)
20         for (var j=0 ; j< myApp.metier.adresse.getAttributeList().length ; ++j){
21             var attributeName = myApp.metier.adresse.getAttributeList()[j];
22             if (attributeName != "id"){

```

```

23 // calcul de l'ID de l'input
24 var inputId = myApp.gui.getInputId({
25     attributeName : attributeName,
26     formId : "ajouterAdresseForm"
27 });
28 // Modification de la propriété de la adresse
29 // avec la valeur saisie dans l'input.
30 nouvelleAdresse.setAttribute(attributeName,
31     document.getElementById(inputId).value
32 );
33 }
34 }
35 if (!nouvelleAdresse.hasError()) {
36     // Provoquer la mise à jour de la vue (panneau des détails)
37     myApp.gui.mediator.publish("personne/detailsChanged", {
38         personne : myApp.modele.selectedPersonne
39     });
40     // Provoquer la requête AJAX pour l'implémentation de la persistance
41     myApp.gui.mediator.publish("adresse/created", {
42         personne : myApp.modele.selectedPersonne,
43         adresse : nouvelleAdresse
44     });
45 } else {
46     myApp.modele.selectedPersonne.deleteAdresse(nouvelleAdresse);
47 }
48 };
49
50 // Enregistrement du callback de l'événement de validation du formulaire
51 myApp.gui.mediator.subscribe("adresse/create", updateModel);
52
53 }()) );

```

### 5.5.4 Modification d'une adresse

La modification d'une adresse après modification présente la difficulté suivante : il faut retrouver l'instance d'adresse à modifier, parmi les adresses de la personne sélectionnée. Nous avons choisi de mettre un champs caché avec l'*ID* dans le formulaire (voir la partie 4.3). Il nous faut alors rechercher l'*ID* de l'adresse dans les instances d'adresse de la personne sélectionnée. Nous aurions aussi pu ajouter une référence vers l'adresse éditée dans le modèle.

Code Source 5.17 : /ihm-demo/guiModifierAdresseFormValidate.js

```

1 /**
2  * Définition et enregistrement du callback réagissant à la validation (submit)
3  * du formulaire de modification d'une personne.
4  */
5 myApp.addModule.apply(myApp.gui, ["callbacksValidateModifierAdresseForm",
6     function() {
7         // Formulaire de modification d'une personne
8
9         /**
10          * Modifie le modèle à partir des données saisies dans le formulaire
11          */
12         var updateModel = function() {

```

```

13 // 1) Mise à jour des données du modèle
14 // à partir des valeurs des inputs du formulaire
15 var attributeName,
16     inputId;
17
18 // Recherche de l'adresse qui a été modifiée à partir de son ID unique
19 // L'ID se trouve en champs caché du formulaire.
20 var inputId_id = myApp.gui.getInputId({
21     attributeName: "id",
22     formId: "modifierAdresseForm"
23 });
24
25 // ID unique de l'adresse concernée par le formulaire
26 var idAdresse = document.getElementById(inputId_id).value;
27 var adresseEnQuestion; // Référence de l'adresse concernée par le formulaire
28 for (var i = 0 ; i < myApp.modele.selectedPersonne.getNbAdresses() ; ++i){
29     if (idAdresse === myApp.modele.selectedPersonne.getAdresse(i).getAttribute(
30         'id')){
31         adresseEnQuestion = myApp.modele.selectedPersonne.getAdresse(i);
32     }
33 }
34 if (adresseEnQuestion === undefined){
35     throw new Error("Adresse introuvable (ID inexistant)");
36 }
37 // On récupère les attributs du formulaire dans une nouvelle instance
38 var changedAdresse = myApp.metier.adresse.createInstance(null);
39 // Pour chaque propriété (chaque input du formulaire)
40 for (var j=0 ; j< myApp.metier.adresse.getAttributeList().length ; ++j){
41     var attributeName = myApp.metier.adresse.getAttributeList()[j];
42     if (attributeName !== "id"){
43         // calcul de l'ID de l'input
44         var inputId = myApp.gui.getInputId({
45             attributeName: attributeName,
46             formId: "modifierAdresseForm"
47         });
48         // Modification de la propriété de la personne
49         // avec la valeur saisie dans l'input.
50         changedAdresse.setAttribute(attributeName,
51             document.getElementById(inputId).value
52         );
53     }
54 }
55 // Seulement s'il n'y a pas d'erreur (filtrage strict côté client)
56 if (!changedAdresse.hasError()){
57     for (var j=0 ; j< myApp.metier.adresse.getAttributeList().length ; ++j){
58         var attributeName = myApp.metier.adresse.getAttributeList()[j];
59         if (attributeName !== "id"){
60             adresseEnQuestion.setAttribute(attributeName,
61                 changedAdresse.getAttribute(attributeName));
62         }
63     }
64     // Provoquer la mise à jour des éléments de la vue observant la personne
65     myApp.gui.mediator.publish("personne/detailsChanged", {
66         personne: null
67     });
68     // Provoquer la mise à jour des éléments de la vue observant la personne

```



```
68     myApp.gui.mediator.publish("adresse/changed", {
69         personne : myApp.modele.selectedPersonne,
70         adresse : adresseEnQuestion
71     });
72 }
73 };
74 // Enregistrement du callback de l'événement de validation du formulaire
75 myApp.gui.mediator.subscribe("adresse/update", updateModel);
76 }()];
```

# Chapitre 6

## Requêtes Asynchrones et *API Restful*

### 6.1 Qu'est-ce qu'une requête asynchrone ?

Les requêtes asynchrones *XMLHttpRequest* permettent d'exécuter (suite à un événement côté client) une requête *HTTP* (exécution d'un script ou programme, par exemple en *PHP*) sur le serveur. On parle de requête *asynchrone* car le client n'est pas bloqué en attendant la réponse du serveur : le déroulement du programme côté client peut se poursuivre, et la réponse du serveur est gérée par des *callbacks*.

Malgré le nom *XMLHttpRequest*, les requêtes asynchrones permettent d'échanger avec le serveur d'autres types de données que du *XML*. Nous utiliserons dans ce cours des données *JSON*.

Le codage *JSON* permet de coder sous forme de chaîne de caractères des collections d'objets. Ainsi, on pourra, par exemple, coder en *JSON* une collection d'objets en *PHP* (tableau associatif), puis transmettre la chaîne *JSON* via une requête asynchrone, et enfin reconstituer une collection d'objets en *JavaScript* pour générer, par exemple, une mise en forme *HTML* dans le document.

Voici un exemple de code *JSON* d'un tableau associatif *PHP* ou d'un objet *JavaScript* (qui contient lui-même un tableau de descriptions de formats) :

Code Source 6.1 : Code *JSON* d'un tableau associatif *PHP* ou objet *JavaScript*

```
1 {  
2   "id": 654,  
3   "denomination": "Tutoriel JavaScript",  
4   "prix unitaire": 0.50,  
5   "formats": [ "PDF", "Postscript", "HTML", "ePub" ]  
6 }
```

On peut, par exemple, générer un tel tableau sur un serveur en *PHP* par le code suivant :

Code Source 6.2 : Génération et sortie du code *JSON* en *PHP*

```
1 <?php  
2 $myArray = array( "id" => 654,  
3                 "denomination" => "Tutoriel JavaScript",  
4                 "prix unitaire" => 0.50,  
5                 "formats" => array( "PDF", "Postscript", "HTML", "ePub" ) );  
6 // Header HTTP spécifique :  
7 header( 'content-type: application/json; charset=utf-8' );  
8 echo json_encode( $myArray );
```

## 6.2 Requ tes *Ajax*

Le m thode `ajax` de *jQuery* permet d'effectuer une requ te *XMLHttpRequest* qui transmet des param tres (un objet *JavaScript*)   un *CGI* (ici en *PHP*), via une *URL*. Dans notre exemple, le serveur re oit lui-m me un objet (propri t  `data`) c d  en *JSON*, et g n re lui-m me du code *JSON*. Le programme client r cup re du code *JSON* g n r e sur la sortie standard du *CGI*, et reconstitue un objet *JavaScript*.

Voici notre exemple o  le code *JavaScript* c t  client r cup re une collection d'objet cr  e par le *CGI* et la met en forme en *HTML*. Trois boutons permettent de tester :

- Un cas sans erreur ;
- Un cas o  la gestion d'erreur est impl ment e en *PHP* c t  serveur ;
- Un cas o  la requ te *AJAX* elle-m me  choue.

Les trois *callbacks* suivants sont utilis s pour g rer la requ te :

- `success` en cas de succ s de la requ te ;
- `error` en cas d' chec de la requ te
- `complete`, ici utilis  pour mettre   jour la vue, que ce soit en cas de succ s ou en cas d' chec de la requ te.

Je programme en *JavaScript* c t  client est le suivant :

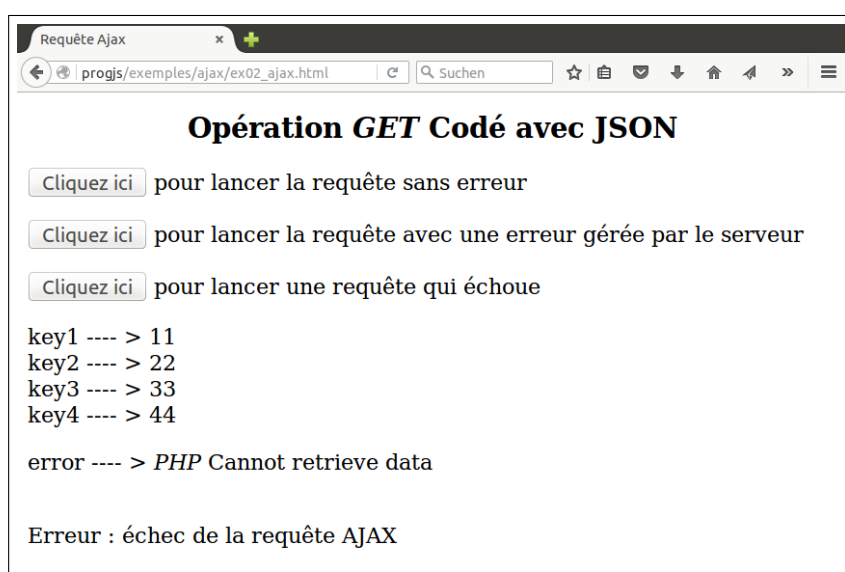


FIGURE 6.1 : Illustration du code source 6.3

Code Source 6.3 : /ajax/ex02-ajax.html (cf. Fig 6.1)

```

1 <!doctype html>
2 <html lang="fr">
3 <head>
4   <meta charset="utf-8">
5   <title>Requête Ajax</title>
6   <script src="/jquery.js"></script>
7   <link rel="stylesheet" href="basicStyle.css"/>
8 </head>
9
10 <body>
11 <h1>Opération <i>GET</i> Codé avec JSON</h1>
12
13 <p><button onclick="lancerRequete(1)">Cliquez ici</button> pour lancer la requête sans erreur</p>
14 <p><button onclick="lancerRequete(0)">Cliquez ici</button> pour lancer la requête avec une erreur gérée par le serveur</p>
15 <p><button onclick="lancerRequete(-1)">Cliquez ici</button> pour lancer une requête qui échoue</p>
16
17 <p id="outputParagraph"></p>
18 <script>
19
20   var model = {
21     paragraphText : "",
22     error : null,
23     getErrorMessage : function() {
24       return this.error !== null ? "<br/>" + this.error : "";
25     }
26   };
27
28   /**
29    * fonction callback exécutée en cas de succès de la requête AJAX.
30    * La méthode parcourt les données retournées par le serveur au format JSON,
31    * et concatène le texte dans le modèle.
32    * @param {Object} retrievedData : collection des données décodées à partir du
33    *                               JSON.
34    *                               La donnée peut être un message d'erreur.
35    */
36   var ajaxCallbackSuccess = function(retrievedData) {
37     model.error=null;
38     model.paragraphText = "";
39     // Parcours et affichage des données de l'objet
40     for (var key in retrievedData) {
41       model.paragraphText += key + " ——— > " + retrievedData[key] + '<br/>';
42     }
43   };
44
45   /**
46    * fonction callback exécutée en cas d'échec de la requête AJAX.
47    * Une erreur est ajoutée dans le modèle et le texte du paragraphe est mis à
48    * vide.
49    */
50   var ajaxCallbackError = function() {
51     model.paragraphText = "";
52     model.error = "Erreur : échec de la requête AJAX";
53   };

```

```
51     };
52
53     /**
54     * fonction callback ex cut e lorsque la requ te AJAX se termine.
55     * Ce callback est appel  en cas d' ch c ET en cas de succ s de la requ te
56     * AJAX.
57     * Ici, la m thode met   jour la vue en affichant le texte et une  ventuelle
58     * erreur.
59     */
60     var ajaxCallbackComplete = function() {
61         $( "#outputParagraph" ).append(
62             "<p>" +
63             model.paragraphText +
64             model.getErrorMessage() +
65             "</p>" );
66     }
67
68     /**
69     * Gestionnaire de click sur les boutons, qui d clenche une requ te AJAX.
70     * @param {int} simpleTestValue donn e transmise au serveur via la propri t 
71     * simpleTest
72     * si simpleTestValue est n gatif, une URL du serveur inexistante
73     * est utilis e,
74     * provoquant l' ch c de la requ te (c'est juste pour l'exemple
75     * ...).
76     */
77     var lancerRequete = function(simpleTestValue){
78
79         var urlServeur = "http://progjs/exemples/ajax/ex01_encode_json.php";
80
81         // Pour provoquer une requ te qui  choue compl tement
82         if (simpleTestValue < 0){ // URL qui n'existe pas
83             urlServeur = "http://progjs/exemples/ajax/bidon.php";
84         }
85
86         // Lancement d'une requ te AJAX avec donn es (POST) cod e en JSON
87         var jqxhr = $.ajax({
88             // Envoyer les donn es de la personne avec le format JSON
89             dataType: "json",
90             url: urlServeur, // URL du serveur
91             method: 'post', // Envoyer les donn es dans le tableau $_POST
92             contentType: 'application/x-www-form-urlencoded',
93             // donn es   transmettre au serveur
94             data : {
95                 simpleTest : simpleTestValue
96             },
97             // M thode callback qui reconstruit le mod le en cas de succ s
98             success : ajaxCallbackSuccess ,
99             // M thode callback qui g re une  ventuelle erreur dans la requ te
100             error : ajaxCallbackError ,
101             // M thode callback qui met   jours la vue la vue en cas de succ s ou d'
102             erreur
103             complete : ajaxCallbackComplete
104         });
105     }
```

```

101 </script>
102 </body>

```

Le programme en *PHP* côté serveur est le suivant :

Code Source 6.4 : /ajax/ex01-encode-json.php

```

1 <?php
2 if (isset($_REQUEST['simpleTest']) && $_REQUEST['simpleTest'] == 1){
3     $myArray = array('key1' => 11, 'key2' => 22, 'key3' => 33, 'key4' => 44);
4 }else{
5     $myArray = array('error' => "<i>PHP</i> Cannot retrieve data");
6 }
7
8 // Header HTTP
9 header('content-type: application/json; charset=utf-8');
10 echo json_encode($myArray);
11 ?>

```

### 6.3 Qu'est-ce qu'une *API REST* (ou systèmes *Restful*) ?

Les explications sur les *Web Services* de type *API Restful* se trouvent dans le cours de programmation *Web* côté serveur sur :

<https://malgouyres.eu/programmation-php>

Nous rappelons ici quelques principes. L'architecture *REST* (*representational state transfer*) est, dans notre cadre, une architecture d'application client-serveur, qui permet le lien entre une application côté client en *Javascript* et un serveur web sur lequel s'exécutent des *CGI*.

Le serveur permettra (au moins) d'effectuer au moins les opérations *CRUD* (*Create, Read, Update, Delete*) sur des instances d'objets *métier*, aussi appelées *entités* ou *ressources* :

- **Opération *Create*.** De créer une ressource (ici une ligne d'une table de base de données) avec ou sans son identifiant unique.

*Exemple 1* : Créer une ressource de type **Adresse** en spécifiant les données de l'adresse, en laissant au serveur le choix de l'*Id* de la ressource créée. Le serveur retourne l'*Id* généré pour que le client le connaisse.

*Exemple 2* : Créer une ressource de type **Adresse** en spécifiant les données de l'adresse **ET** l'identifiant unique de l'instance à créer, par exemple parce que cet *Id* doit être généré par un algorithme dépendant du client, ou parce que cet *Id* doit correspondre à l'*Id* de la même entité ailleurs sur le réseau (comme l'*ISBN* d'un livre, qui ne peut pas être choisi au hasard par le serveur).

- **Opération *Read*.** De lire toutes les ressources (ici d'une table de base de données).  
*Exemple* : Lire toutes les personnes de la table **Personne**, avec une collection d'adresses pour chaque personne (résultat d'une jointure en *SQL* qui correspond à un agrégat sur les objets métiers).
- **Opération *Read* avec *Id* ou prédicat.** De lire **ou bien** une ressource identifiée de manière unique par un identifiant unique (une ligne d'une table de base de données) **ou bien** un certain nombre de ressources données par le résultat d'une requête (comme un *SELECT* en *SQL*) ou par les données d'une jointure (par exemple avec l'identifiant d'un

agrégat).

*Exemple 1* : Lire l'adresse d'identifiant unique (clé primaire de la table **Adresse** égal à **af49bc053de73a0**).

*Exemple 2* : Lire toutes les adresses de la personne d'identifiant unique **bd56bc053de12b3**.

*Exemple 3* : Lire toutes les personnes de la table **Personne** qui ont une adresse avec le code postal commençant par les deux chiffre **63**.

- **Opération *Update***. De mettre à jour une ressource (ici une ligne d'une table de base de données) identifiée de manière unique (par un identifiant unique), avec des données (partielles ou complètes) à modifier.

*Exemple* : Modifier le code postal d'une adresse d'identifiant unique égal à **af49bc053de73a0**.

- **Opération *Delete***. De détruire une ressource (ici une ligne d'une table de base de données) identifiée de manière unique (par un identifiant unique);

*Exemple* : Détruire la personne d'identifiant unique **bd56bc053de12b3**, ainsi que (s'agissant d'une composition) toutes ses adresses de la table **adresse** (utilisation d'une clé étrangère).

En utilisant cette interface (*service web*), l'application côté client pourra accéder à la couche persistance du serveur.

Nous voyons aussi comment implémenter ces opérations sur le serveur en spécifiant les identifiants et les actions au moyen d'une *URI* (*Universal Ressource Identifier*) et des *verbes* (aussi appelés *méthodes*, GET, PUT, POST, PATCH ou DELETE du protocole *HTTP* (norme *RFC 2616* puis *RFC 7230*).

## 6.4 Persistance par Requêtes sur une *API Restful*

### 6.4.1 Création du Module persistance et Objet statusCode

À la racine d'un nouveau module **myApp.persistance**, nous créons un objet **statusCodeObject**, destiné à définir les méthodes *callback* de gestion d'un retour de code d'erreur (*status code*) *HTTP*.

Code Source 6.5 : /clientAndAPI/client/persistanceCommon.js

```
1  /**
2   * Définition du module chargé de la persistance par requêtes AJAX
3   * sur une API Restful permettant d'accéder à des entités et de
4   * stocker des entités.
5   */
6  myApp.addModule( "persistance", {} );
7
8  /**
9   * Définition et de l'objet définissant les méthode callback
10  * correspondant aux différents codes d'erreur (status codes) HTTP.
11  */
12  myApp.addModule.apply( myApp.persistance , [ "statusCodeObject", {
13    404 : function() {
14      alert( "Resource not found" );
15    },
16    400 : function() {
17      alert( "Bad Request" );
```

```

18     },
19     405 : function() {
20         alert("Method Not Allowed");
21     },
22     422 : function() {
23         alert("Unprocessable Entity : Attribut incorrect ?");
24     },
25     500 : function() {
26         alert("Internal Server Error");
27     }
28 }));

```

### 6.4.2 Construction du modèle à partir de la base de données

Code Source 6.6 : /clientAndAPI/client/persistenceRead.js

```

1  /**
2   * Définition et enregistrement des callbacks de chargement du modèle
3   * à partir des données sur le serveur par une requête AJAX.
4   * Permet le chargement du modèle à partir de la base de données.
5   */
6  myApp.addModule.apply(myApp.persistance, [ "callbacksRebuildModelFromServer",
7      function() {
8          /**
9           * Méthode callback qui est appelée en cas de succès de la requête AJAX.
10          * Cette méthode reconstruit le modèle à partir des données du serveur.
11          * @param {Object} retrievedData données reçues du serveur (après parsing du
12             JSON)
13          * @param {Object|null} retrievedData.error null en l'absence d'erreur détectée
14             e par le serveur,
15          * ou un objet dont les propriétés sont les messages d'erreur renvoyés
16             es par le serveur.
17          * @param {Object} retrievedData.data données renvoyées par le serveur :
18          * collections d'objets permettant de construire des personnes
19             , avec leurs adresses.
20          */
21          var ajaxCallbackSuccess = function(retrievedData){
22              var adressesData, adresseInstance;
23
24              // Si aucune erreur n'a été détectée sur le serveur
25              if (retrievedData["error"] === null && retrievedData['data'] !== undefined){
26                  // Parcours des objets dans les données
27                  for (var key in retrievedData['data']){
28                      if (retrievedData['data'].hasOwnProperty(key)){
29                          // Création d'une personne sans adresse
30                          var newPersonne = myApp.metier.personne.createInstance({
31                              id : retrievedData['data'][key]["id"],
32                              nom : retrievedData['data'][key]["nom"]
33                          });
34
35                          // Parcours des objets définissant les adresses
36                          adressesData = retrievedData['data'][key]["adresses"];
37                          for (var keyAdresse in adressesData){
38                              if (adressesData.hasOwnProperty(keyAdresse)){
39                                  // Création et ajout d'une adresse

```



```

35         adresseInstance = myApp.metier.adresse.createInstance({
36             id : adressesData[keyAdresse][ "id" ],
37             numeroRue : adressesData[keyAdresse][ "numeroRue" ],
38             rue : adressesData[keyAdresse][ "rue" ],
39             complementAddr : adressesData[keyAdresse][ "complementAddr" ],
40             codePostal : adressesData[keyAdresse][ "codePostal" ],
41             ville : adressesData[keyAdresse][ "ville" ],
42             pays : adressesData[keyAdresse][ "pays" ]
43         });
44         newPersonne.addAdresse(adresseInstance);
45     }
46 }
47 myApp.modele.personnes.push(newPersonne); // ajout dans le modèle
48 }
49 }
50 }else{
51     alert("Il y a un objet \"error\" non null");
52 }
53 };
54
55 /**
56  * Méthode appelée lorsque la requête AJAX se termine,
57  * que ce soit après une erreur ou après un succès.
58  * Cette méthode reconstruit la vue (après reconstruction du modèle).
59  */
60 var ajaxCallbackComplete = function(retrievedData){
61
62     // Personne sélectionnée par défaut
63     myApp.modele.selectedPersonne = myApp.modele.personnes[0];
64
65     // La vue est réinitialisée : on vide les éléments et événements
66     $("#listePersonnes").empty();
67     $("#vueDetail").empty();
68     $("#ajouterPersonneForm").empty();
69     $("#modifierPersonneForm").empty();
70     $("#ajouterAdresseForm").empty();
71     $("#modifierAdresseForm").empty();
72
73     // Provoquer le premier affichage de la vue :
74     myApp.gui.mediator.publish("personne/changed", {
75         personne : myApp.modele.selectedPersonne
76     });
77
78     // Enregistrement des événements utilisateurs gérés par jQuery
79     myApp.gui.initJQueryEventsPersonne();
80     myApp.gui.initJQueryEventsAdresse();
81 };
82
83 /**
84  * Callback appelé lors de l'événement "personne/read" du médiateur.
85  * Effectue une requête AJAX pour récupérer toutes les personnes
86  * pour reconstruire le modèle de données.
87  */
88 var readAllPersonne = function(){
89     // requête AJAX get codé en JSON
90     var jqxhr = $.ajax({

```

```

91 // Envoyer les données de la personne avec le format JSON
92 dataType : "json",
93 url : "http://progjs/exemples/apiRestful/personne", // URL du serveur
94 method : 'get', // Verbe HTTP
95 contentType : 'application/x-www-form-urlencoded',
96 // données à transmettre au serveur
97 data : {
98     //action : "personne-get-all"
99 },
100 // Méthode callback qui reconstruit le modèle en cas de succès
101 success : ajaxCallbackSuccess,
102 // Méthode callback qui gère une éventuelle erreur dans la requête
103 complete : ajaxCallbackComplete,
104 // Objet définissant les callbacks d'erreurs avec codes du serveur
105 statusCode : myApp.persistance.statusCodeObject
106 });
107 };
108
109 // Enregistrement du callback de l'événement de reconstruction du modèle
110 myApp.gui.mediator.subscribe("personne/read", readAllPersonne);
111 }());

```

Code Source 6.7 : /clientAndAPI/client/index.html

```

1 <!doctype HTML>
2 <html lang="fr">
3 <head>
4     <meta charset="UTF-8" />
5     <title>Application interactive</title>
6     <link rel="stylesheet" href="basicStyle.css"/>
7 </head>
8 <body>
9     <!-- Structure de l'application vide avec deux méthodes -->
10    <script src="../../pattern-fonct/ex04-structureApplication.js"></script>
11    <!-- Création de sous-module regexUtil de myApp.metier -->
12    <script src="../../pattern-fonct/ex05-modulePatternRegex.js"></script>
13    <!-- Sous-module adresse de myApp.metier -->
14    <script src="../../pattern-proto/ex05-createModuleMetierProto.js"></script>
15    <!-- Méthode fabrique générique d'objets métier -->
16    <script src="../../pattern-proto/ex05-fabriqueObjetMetierProto.js"></script>
17    <!-- Création de sous-module adresse de myApp.metier -->
18    <script src="../../pattern-fonct/ex06-moduleMetierAdresse.js"></script>
19    <!-- Création d'une méthode fabrique d'adresse de myApp.metier.adresse -->
20    <script src="../../pattern-fonct/ex08-fabriqueAdresse.js"></script>
21    <!-- Classe de vérification de l'implémentation d'interfaces -->
22    <script src="../../pattern-fonct/ex11-interfaceImplementation.js"></script>
23    <!-- Classe de vérification de l'implémentation d'interfaces -->
24    <script src="../../pattern-fonct/ex11-interfaceImplementationMetier.js"></s
    cript>
25    <!-- Création de fonctions d'affichage dans myApp.metier.view.adresse -->
26    <script src="../../pattern-fonct/ex09-adresseView.js"></script>
27
28
29
30    <!-- Mediator spécialisé pour filtrer les inputs (evt "change") -->
31    <script src="../../form-filter/ex02-mediatorInputFilter.js"></script>

```

```

32 <!-- Génération automatique de formulaires avec filtrage des attributs -->
33 <script src="../../form-filter/ex03-formsGui.js"></script>
34
35 <!-- Module Métier myApp.metier.personnes avec collection getAdresses() -->
36 <script src="../../ihm-demo/personneModule.js"></script>
37 <!-- Construction en dur d'un modèle de données : collection de personnes -->
38 <script src="../../ihm-demo/modelModule.js"></script>
39 <!-- Pattern Médiateur pour enchainement architecture trois tiers -->
40 <script src="../../ihm-demo/mediator.js"></script>
41
42 <!-- Inclusion de jQuery pour les événements et manipulation du DOM -->
43 <script src="jquery.js"></script>
44 <!-- Événements utilisateurs concernant les personnes -->
45 <script src="../../ihm-demo/guiJQueryEventsPersonne.js"></script>
46 <!-- Événements utilisateurs concernant les adresses -->
47 <script src="../../ihm-demo/guiJQueryEventsAdresse.js"></script>
48
49 <!-- Implémentation de l'action "click sur 'Modifier' de Personne" -->
50 <script src="../../ihm-demo/guiBoutonModifierPersonne.js"></script>
51 <!-- Implémentation de l'action "click sur 'Supprimer' de Personne" -->
52 <script src="../../ihm-demo/guiBoutonSupprimerPersonne.js"></script>
53 <!-- Implémentation de l'action "click sur 'Ajouter' de Personne" -->
54 <script src="../../ihm-demo/guiBoutonAjouterPersonne.js"></script>
55 <!-- Implémentation de l'action "click sur 'Ajouter' de Adresse" -->
56 <script src="../../ihm-demo/guiBoutonAjouterAdresse.js"></script>
57 <!-- Implémentation de l'action "click sur 'Ajouter' de Adresse" -->
58 <script src="../../ihm-demo/guiBoutonModifierAdresse.js"></script>
59 <!-- Implémentation de l'action "click sur 'Supprimer' de Adresse" -->
60 <script src="../../ihm-demo/guiBoutonSupprimerAdresse.js"></script>
61 <!-- Implémentation de l'action "Validation de formulaire de modif" -->
62 <script src="../../ihm-demo/guiModifierPersonneFormValide.js"></script>
63 <!-- Implémentation de l'action "Validation de formulaire d'ajout" -->
64 <script src="../../ihm-demo/guiAjouterPersonneFormValide.js"></script>
65 <!-- Implémentation de l'action "Validation de formulaire de modif" -->
66 <script src="../../ihm-demo/guiModifierAdresseFormValide.js"></script>
67 <!-- Implémentation de l'action "Validation de formulaire d'ajout" -->
68 <script src="../../ihm-demo/guiAjouterAdresseFormValide.js"></script>
69 <!-- Mise à jour de la vue (panneau "détails" uniquement) -->
70 <script src="../../ihm-demo/guiDetailsChanged.js"></script>
71 <!-- Mise à jour de la vue (re-générer toute la vue) -->
72 <script src="../../ihm-demo/guiPersonneChanged.js"></script>
73
74 <script src="/persistanceCommon.js"></script>
75 <script src="/persistanceRead.js"></script>
76 <script src="/persistanceCreatePersonne.js"></script>
77 <script src="/persistanceDeletePersonne.js"></script>
78 <script src="/persistanceUpdatePersonne.js"></script>
79 <script src="/persistanceCreateAdresse.js"></script>
80 <script src="/persistanceUpdateAdresse.js"></script>
81 <script src="/persistanceDeleteAdresse.js"></script>
82 <!-- Code HTML de la vue -- Structure générale de la page HTML -->
83
84 <button id="boutonAjouterPersonne">Ajouter une personne</button><br/>
85 <span id="listePersonnes" class="panel"></span>
86 <span class="panel">
87   <span id="vueDetail">

```

```

88     </span><br /><br />
89 </span>
90 <span id="spanMainForm" class="panel">
91     <form id="ajouterPersonneForm" method="post" >/form>
92     <form id="modifierPersonneForm" method="post" >/form>
93     <form id="ajouterAdresseForm" method="post" >/form>
94     <form id="modifierAdresseForm" method="post" >/form>
95 </span>
96
97 <!-- Ajout d'un main et exécution -->
98 <script>
99     /**
100      * Série d'instructions effectuées pour initialiser l'application/
101      * @method mainFunction
102      * @augments myApp
103      */
104     myApp.addModule( "mainFunction", function() {
105
106         // Création d'un modèle avec une collection de Personne vide
107         myApp.addModule.apply(myApp, [ "modele", {
108             selectedPersonne : null,
109             personnes : [],
110         }]);
111         // Charger le modèle :
112         myApp.gui.mediator.publish( "personne/read", {
113             personne : myApp.modele.selectedPersonne
114         });
115     });
116
117     //////////////////////////////////////
118     // Exécution de la méthode mainFunction
119     myApp.mainFunction();
120 </script>
121 </body>
122 </html>

```

### 6.4.3 Création, Mise à jour, et suppression des personnes

Code Source 6.8 : /clientAndAPI/client/persistenceCreatePersonne.js

```

1  /**
2   * Définition et enregistrement des callbacks de création d'une personne
3   * sur le serveur par requête AJAX.
4   */
5  myApp.addModule.apply(myApp.persistance, [ "callbacksCreatePersonneQueryServer",
6      function() {
7          /**
8           * Méthode callback qui est appelée en cas de succès de la requête AJAX.
9           * Cette méthode informe simplement l'utilisateur des éventuelles erreurs.
10          * En effet, la requête n'est pas supposée retourner des données.
11          */
12          var ajaxCallbackSuccess = function(retrievedData) {
13              var concatErrorMsg="";
14              if (retrievedData["error"] !== null) {
15                  for (var key in retrievedData['error']) {

```

```

15         if (retrievedData[ 'error' ].hasOwnProperty(key)){
16             concatErrorMsg += key + ": " + retrievedData[ 'error' ][key] + "\n";
17         }
18     }
19     alert (concatErrorMsg);
20 }
21 };
22
23 /**
24  * Callback appelé lors de l'événement "personne/read" du médiateur.
25  * Effectue une requête AJAX pour récupérer toutes les personnes
26  * pour reconstruire le modèle de données.
27  */
28 var createPersonne = function(contextArg){
29
30     // requête AJAX get codé en JSON
31     var jqxhr = $.ajax({
32         dataType: "json", // On envoie les données la personne codée en JSON
33         // L'URI complète inclus l'ID de la ressource à créer
34         url: "http://progjs/exemples/apiRestful/personne/"
35             + contextArg.personne.getAttribute("id"),
36         method: 'post', // Verbe HTTP
37         contentType: 'application/x-www-form-urlencoded',
38         // données à transmettre au serveur
39         data : {
40             personne: { // Attributs de la personne
41                 nom: contextArg.personne.getAttribute("nom")
42             }
43         },
44         // Méthode callback qui reconstruit le modèle en cas de succès
45         success: ajaxCallbackSuccess,
46         // Objet définissant les callbacks d'erreurs avec codes du serveur
47         statusCode: myApp.persistance.statusCodeObject
48     });
49 };
50 // Enregistrement du callback de l'événement de mise à jour de la personne
51 myApp.gui.mediator.subscribe("personne/created", createPersonne);
52 }());

```

Code Source 6.9 : /clientAndAPI/client/persistanceUpdatePersonne.js

```

1  /**
2   * Définition et enregistrement des callbacks de modification d'une personne
3   * sur le serveur par requête AJAX.
4   */
5  myApp.addModule.apply(myApp.persistance, [ "callbacksUpdatePersonneQueryServer",
6      function() {
7          /**
8           * Méthode callback qui est appelée en cas de succès de la requête AJAX.
9           * Cette méthode informe simplement l'utilisateur des éventuelles erreurs.
10          * En effet, la requête n'est pas supposée retourner des données.
11          */
12          var ajaxCallbackSuccess = function(retrievedData){
13              var concatErrorMsg="";
14              if (retrievedData["error"] !== null){
15                  for (var key in retrievedData[ 'error' ]){

```

```

15         if (retrievedData[ 'error' ].hasOwnProperty(key)){
16             concatErrorMsg += key + ": " + retrievedData[ 'error' ][key] + "\n";
17         }
18     }
19     alert (concatErrorMsg);
20 }
21 };
22
23 /**
24  * Callback appelé lors de l'événement "personne/read" du médiateur.
25  * Effectue une requête AJAX pour récupérer toutes les personnes
26  * pour reconstruire le modèle de données.
27  */
28 var updatePersonne = function(contextArg){
29
30     // requête AJAX get codé en JSON
31     var jqxhr = $.ajax({
32         dataType: "json", // On envoie les données la personne codée en JSON
33         // L'URI complète inclus l'ID de la ressource à créer
34         url: "http://progjs/exemples/apiRestful/personne/"
35             + contextArg.personne.getAttribute("id"),
36         method: 'put', // Verbe HTTP
37         contentType: 'application/x-www-form-urlencoded',
38         // données à transmettre au serveur
39         data: {
40             personne: { // Attributs de la personne
41                 nom: contextArg.personne.getAttribute("nom")
42             }
43         },
44         // Méthode callback qui reconstruit le modèle en cas de succès
45         success: ajaxCallbackSuccess,
46         // Objet définissant les callbacks d'erreurs avec codes du serveur
47         statusCode: myApp.persistance.statusCodeObject
48     });
49 };
50 // Enregistrement du callback de l'événement de mise à jour de la personne
51 myApp.gui.mediator.subscribe("personne/changed", updatePersonne);
52 }());

```

Code Source 6.10 : /clientAndAPI/client/persistanceDeletePersonne.js

```

1 /**
2  * Définition et enregistrement des callbacks de suppression d'une personne
3  * sur le serveur par requête AJAX.
4  */
5 myApp.addModule.apply(myApp.persistance, ["callbacksDeletePersonneQueryServer",
6     function(){
7         /**
8          * Callback appelé lors de l'événement "personne/read" du médiateur.
9          * Effectue une requête AJAX pour récupérer toutes les personnes
10         * pour reconstruire le modèle de données.
11         */
12         var deletePersonne = function(contextArg){
13
14             // requête AJAX get codé en JSON
15             var jqxhr = $.ajax({

```

```

15     dataType : "json", // On envoie les données la personne codée en JSON
16     // L'URI complète inclus l'ID de la ressource à supprimer
17     url : "http://progjs/exemples/apiRestful/personne/"
18           + contextArg.personne.getAttribute("id"),
19     method : 'delete', // Verbe HTTP
20     contentType : 'application/x-www-form-urlencoded',
21     // données à transmettre au serveur
22     data : {
23         // Objet vide (le paramètre ID est spécifié dans l'URI)
24     },
25     // Objet définissant les callbacks d'erreurs avec codes du serveur
26     statusCode : myApp.persistance.statusCodeObject
27 });
28
29 };
30 // Enregistrement du callback de l'événement de mise à jour de la personne
31 myApp.gui.mediator.subscribe("personne/delete", deletePersonne);
32 }());

```

#### 6.4.4 Création, Mise à jour, et suppression des adresses

Code Source 6.11 : /clientAndAPI/client/persistanceCreateAdresse.js

```

1  /**
2   * Définition et enregistrement des callbacks de création d'une personne
3   * sur le serveur par requête AJAX.
4   */
5  myApp.addModule.apply(myApp.persistance, ["callbacksCreateAdresseQueryServer",
6      function() {
7          /**
8           * Méthode callback qui est appelée en cas de succès de la requête AJAX.
9           * Cette méthode informe simplement l'utilisateur des éventuelles erreurs.
10          * En effet, la requête n'est pas supposée retourner des données.
11          */
12          var ajaxCallbackSuccess = function(retrievedData){
13              var concatErrorMsg="";
14              if (retrievedData["error"] !== null){
15                  for (var key in retrievedData['error']){
16                      if (retrievedData['error'].hasOwnProperty(key)){
17                          concatErrorMsg += key + ": " + retrievedData['error'][key] + "\n";
18                      }
19                  }
20                  alert(concatErrorMsg);
21              }
22          };
23
24          /**
25           * Callback appelé lors de l'événement "personne/read" du médiateur.
26           * Effectue une requête AJAX pour récupérer toutes les personnes
27           * pour reconstruire le modèle de données.
28           */
29          var createAdresse = function(contextArg){
30              // requête AJAX get codé en JSON
31              var jqxhr = $.ajax({

```

```

32     dataType : "json", // On envoie les données la personne codée en JSON
33     // L'URI complète inclus l'ID de la ressource à créer
34     url : "http://progjs/exemples/apiRestful/adresse/"
35           + contextArg.adresse.getAttribute("id"),
36     method : 'post', // Verbe HTTP
37     contentType : 'application/x-www-form-urlencoded',
38     // données à transmettre au serveur
39     data : {
40         adresse : { // Attributs de l'adresse
41             idPersonne : contextArg.personne.getAttribute("id"),
42             numeroRue : contextArg.adresse.getAttribute("numeroRue"),
43             rue : contextArg.adresse.getAttribute("rue"),
44             complementAddr : contextArg.adresse.getAttribute("complementAddr"),
45             codePostal : contextArg.adresse.getAttribute("codePostal"),
46             ville : contextArg.adresse.getAttribute("ville"),
47             pays : contextArg.adresse.getAttribute("pays")
48         }
49     },
50     // Méthode callback qui reconstruit le modèle en cas de succès
51     success : ajaxCallbackSuccess,
52     // Objet définissant les callbacks d'erreurs avec codes du serveur
53     statusCode : myApp.persistance.statusCodeObject
54 });
55 };
56 // Enregistrement du callback de l'événement de mise à jour de la personne
57 myApp.gui.mediator.subscribe("adresse/created", createAdresse);
58 }());

```

Code Source 6.12 : /clientAndAPI/client/persistanceUpdateAdresse.js

```

1  /**
2   * Définition et enregistrement des callbacks de création d'une personne
3   * sur le serveur par requête AJAX.
4   */
5  myApp.addModule.apply(myApp.persistance, ["callbacksUpdateAdresseQueryServer",
6      function() {
7      /**
8       * Méthode callback qui est appelée en cas de succès de la requête AJAX.
9       * Cette méthode informe simplement l'utilisateur des éventuelles erreurs.
10      * En effet, la requête n'est pas supposée retourner des données.
11      */
12      var ajaxCallbackSuccess = function(retrievedData){
13          var concatErrorMsg="";
14          if (retrievedData["error"] !== null){
15              for (var key in retrievedData['error']){
16                  if (retrievedData['error'].hasOwnProperty(key)){
17                      concatErrorMsg += key + ": " + retrievedData['error'][key] + "\n";
18                  }
19              }
20              alert (concatErrorMsg);
21          }
22      };
23      /**
24       * Callback appelé lors de l'événement "personne/read" du médiateur.
25       * Effectue une requête AJAX pour récupérer toutes les personnes

```



```

26  * pour reconstruire le mod le de donn es.
27  */
28  var updateAdresse = function(contextArg){
29
30      // requ te AJAX get cod  en JSON
31      var jqxhr = $.ajax({
32          dataType: "json", // On envoie les donn es la personne cod e en JSON
33          // L'URI compl te inclus l'ID de la ressource   mettre   jour
34          url: "http://progjs/exemples/apiRestful/adresse/"
35              + contextArg.adresse.getAttribute("id"),
36          method: 'put', // Verbe HTTP
37          contentType: 'application/x-www-form-urlencoded',
38          // donn es   transmettre au serveur
39          data: {
40              adresse: { // Attributs de l'adresse
41                  idPersonne : contextArg.personne.getAttribute("id"),
42                  numeroRue: contextArg.adresse.getAttribute("numeroRue"),
43                  rue: contextArg.adresse.getAttribute("rue"),
44                  complementAddr: contextArg.adresse.getAttribute("complementAddr"),
45                  codePostal: contextArg.adresse.getAttribute("codePostal"),
46                  ville: contextArg.adresse.getAttribute("ville"),
47                  pays: contextArg.adresse.getAttribute("pays")
48              }
49          },
50          // M thode callback qui reconstruit le mod le en cas de succ s
51          success: ajaxCallbackSuccess,
52          // Objet d finissant les callbacks d'erreurs avec codes du serveur
53          statusCode: myApp.persistance.statusCodeObject
54      });
55  };
56  // Enregistrement du callback de l' v nement de mise   jour de la personne
57  myApp.gui.mediator.subscribe("adresse/changed", updateAdresse);
58  }());

```

Code Source 6.13 : /clientAndAPI/client/persistanceDeleteAdresse.js

```

1  /**
2   * D finition et enregistrement des callbacks de cr ation d'une personne
3   * sur le serveur par requ te AJAX.
4   */
5  myApp.addModule.apply(myApp.persistance, ["callbacksDeleteAdresseQueryServer",
6      function(){
7          /**
8           * M thode callback qui est appel e en cas de succ s de la requ te AJAX.
9           * Cette m thode informe simplement l'utilisateur des  ventuelles erreurs.
10          * En effet, la requ te n'est pas suppos e retourner des donn es.
11          */
12          var ajaxCallbackSuccess = function(retrievedData){
13              var concatErrorMsg="";
14              if (retrievedData["error"] !== null){
15                  for (var key in retrievedData['error']){
16                      if (retrievedData['error'].hasOwnProperty(key)){
17                          concatErrorMsg += key + ": " + retrievedData['error'][key] + "\n";
18                      }
19                  }
20              }
21              alert(concatErrorMsg);
22          }
23      }
24  );

```

```
20     }
21   };
22
23   /**
24    * Callback appelé lors de l'événement "personne/read" du médiateur.
25    * Effectue une requête AJAX pour récupérer toutes les personnes
26    * pour reconstruire le modèle de données.
27    */
28   var deleteAdresse = function(contextArg){
29
30     // requête AJAX get codé en JSON
31     var jqxhr = $.ajax({
32       dataType: "json", // On envoie les données la personne codée en JSON
33       // L'URI complète inclus l'ID de la ressource à supprimer
34       url: "http://progjs/exemples/apiRestful/adresse/"
35           + contextArg.adresse.getAttribute("id"),
36       method: 'delete', // Verbe HTTP
37       contentType: 'application/x-www-form-urlencoded',
38       // données à transmettre au serveur
39       data : {
40         // Objet vide (le paramètre ID est spécifié dans l'URI)
41       },
42       // Méthode callback qui reconstruit le modèle en cas de succès
43       success: ajaxCallbackSuccess,
44       // Objet définissant les callbacks d'erreurs avec codes du serveur
45       statusCode: myApp.persistance.statusCodeObject
46     });
47
48   };
49   // Enregistrement du callback de l'événement de mise à jour de la personne
50   myApp.gui.mediator.subscribe("adresse/delete", deleteAdresse);
51 }());
```

# Annexe A

## Graphisme avec les Canvas *HTML5*

### A.1 Notion de *canvas*

Les *canvas HTML5* fournissent une petite *API* graphique 2D en *javascript* qui permet de réaliser des dessins, des graphiques, etc. sans plugin. Les canvas 2D sont dorés et déjà disponible sur tous les grands navigateurs. L'extension *webGL* (qui dépasse le cadre de ce cours) permet de faire des affichage de scènes 3D en accédant aux fonctionnalités d'*OpenGL* via les shaders en *GLSL*. L'extension *webGL* est implémentée dans tous les Grands Navigateurs mais n'est pas implémentée à ce jour dans *internet explorer* car l'éditeur de ce navigateur préfère privilégier une solution propriétaire.

Voici un exemple avec un canvas qui dessine un triangle.

Code Source A.1 : Génération et sortie du code *JSON* en *PHP*

```
1 <!doctype HTML>
2 <html lang="fr">
3   <head>
4     <meta charset="UTF-8" />
5     <title>Mon premier canvas HTML5</title>
6   </head>
7   <body>
8     <!-- Déclaration d'un canvas vide avec son id -->
9     <canvas id="monCanvas" width="2000" height="1000" style="position : absolute;"
10       </canvas>
11     <script>
12       // On récupère le canvas pour dessiner
13       var myCanvas = document.getElementById( "monCanvas" );
14       // On récupère un contexte du canvas pour utiliser les méthodes de
15       // dessin
16       var context = myCanvas.getContext( "2d" );
17       // couleur de remplissage rouge
18       context.fillStyle = "#FF0000";
19
20       context.beginPath();
21       context.moveTo(10, 10);
22       context.lineTo(100, 100);
23       context.lineTo(190, 10);
24       context.lineTo(10, 10);
25
26       context.fill();
27       context.closePath();
```

```
26     </script>
27     <h1>Page HTML avec un canvas</h1>
28     <p>
29     </p>
30 </body>
31 </html>
```

## A.2 Exemple d'animation dans un *canvas*

Voici un exemple qui réalise une animation à l'aide d'un timer qui exécute la fonction `animate` toutes les *20ms*, soit 50 fois par seconde.

Code Source A.2 : Génération et sortie du code *JSON* en *PHP*

```
1 <!doctype HTML>
2 <html lang="fr">
3   <head>
4     <meta charset="UTF-8" />
5     <title>Mon premier canvas HTML5</title>
6   </head>
7   <body>
8     <!-- Déclaration d'un canvas vide avec son id -->
9     <canvas id="monCanvas" width="2000" height="1000" style="position : absolute ;">
10       </canvas>
11     <script>
12       var timer = setInterval(animate, 20);
13
14       function animate() {
15
16         // On récupère le canvas pour dessiner
17         var canvas = document.getElementById( "monCanvas" );
18         // On récupère un contexte du canvas pour utiliser les méthodes de
19         // dessin
20         var context = canvas.getContext( "2d" );
21         // couleur de remplissage rouge
22         context.fillStyle = "#FF0000";
23         context.beginPath();
24         var d = new Date();
25         var n = d.getTime();
26         // nombre de millisecondes depuis le 01/01/1970
27
28         var sec = n / 1000.0;
29         context.clearRect ( 0 , 0 , canvas.width , canvas.height );
30
31         context.save();
32         context.translate(200+500 * (1+Math.cos(0.5 * sec)), 200+200 *(1.0+
33           Math.sin(sec)));
34         // l'angle de rotation doit être entre 0 et 2*Math.PI'
35         context.rotate(sec - 2*Math.PI*Math.round (sec/(2*Math.PI)));
36         context.moveTo(0, 0);
37         context.lineTo(100, 100);
38         context.lineTo(200, 0);
39         context.lineTo(0, 0);
40
41         context.fill();
```

```
39         context.closePath();
40         context.restore();
41     }
42     </script>
43     <h1>Page HIML avec un canvas</h1>
44     <p>
45
46     </p>
47 </body>
48 </html>
```

## Annexe B

# Programmation Événementielle en *JavaScript*

### B.1 Rappel sur la Gestion d'Événements en *CSS*

Dans un style *CSS*, on peut mettre des styles différents sur une balise *HTML* donnée, suivant le contexte utilisateur, via la notion d'événement. Dans l'exemple suivant, le style d'un lien est modifié suivant que le lien a déjà été cliqué, ou si la souris survole le lien (événement *hover*).

Code Source B.1 : Génération et sortie du code *JSON* en *PHP*

```
1  /* style par défaut des liens */
2  a :link {
3      text-decoration : none ;
4      color : #00e ; /* bleu clair */
5  }
6  /* style des liens visités */
7  a :visited {
8      text-decoration : none ;
9      color : #c0c ; /* mauve */
10 }
11 /* style des liens visités */
12 a :hover {
13     text-decoration : underline ; /* souligné */
14     color : #e40 ; /* rouge vif */
15 }
```

Voici un autre exemple, dans lequel un élément *HTML* (ici une balise `<span>` et son contenu) apparaît en *popup* pour afficher les détails d'une personne lors du survol de nom de la personne.

La balise *span* (au sein d'un paragraphe d'une classe *CSS* spécifique appelé `popupDetails`) est par défaut invisible (propriété `display` à `none`). Cette même balise *span* devient visible lorsque le paragraphe est survolé.

Code Source B.2 : Génération et sortie du code *JSON* en *PHP*

```
1  <!doctype html>
2  <html lang="fr">
3      <head>
4          <meta charset="utf-8"/>
5          <link rel="stylesheet" href="./myStyle.css"/>
6          <style>
```

```

7      body{
8          font-family : "Comic Sans MS";
9          font-size : 120%;
10     }
11     h1{
12         margin : 0 auto;
13         text-align : center;
14     }
15     p.popupDetails{
16         background-color : yellow;
17         position : relative; /* pour positioner le span en absolu */
18         max-width : 200px;
19     }
20     p.popupDetails span {
21         display : none;
22     }
23     p.popupDetails:hover span {
24         position : absolute;
25         left : 200px;
26         top : -30;
27         min-width : 500px;
28         background-color : black;
29         color : white;
30         border-radius : 20px;
31         padding : 10px;
32         display : block;
33     }
34 </style>
35 <title>Popups en HTML et CSS</title>
36 </head>
37 <body>
38     <!-- début du corps HTML -->
39     <h1><i>Popup</i> en <i>HTML</i> et <i>CSS</i></h1>
40     <p class="popupDetails">
41         Scarlett Johansson
42         <span>née le 22 novembre 1984 à New York ,
43             est une actrice et chanteuse américaine.<br/>
44             (source&nbsp;: wikipédia)
45         </span>
46     </p>
47 </body>
48 <!-- fin du corps HTML -->
49 </html>
50 <!-- fin du code HTML -->

```

## B.2 Événements en *Javascript*

### B.2.1 Le principe des événements en *Javascript*

Les événements en *Javascript* permettent, en réponse à un événement sur un élément *HTML* du document, d'appeler une fonction *callback* en *Javascript*. Ceci suffit à créer une interface homme machine (*IHM*) côté client, basée sur de la programmation événementielle en *Javascript*.

Une liste (non exhaustive; Voir sur le *web* pour la liste complète)

### 1. Événements souris

- (a) `onclick` : sur un simple clic
- (b) `ondblclick` : sur un double clic
- (c) `onmousedown` : lorsque le bouton de la souris est enfoncé, sans forcément le relâcher
- (d) `onmousemove` : lorsque la souris est déplacée
- (e) `onmouseout` : lorsque la souris sort de l'élément
- (f) `onmouseover` : lorsque la souris est sur l'élément
- (g) `onmouseup` : lorsque le bouton de la souris est relâché

### 2. Événements clavier

- (a) `onkeydown` : lorsqu'une touche est enfoncée
- (b) `onkeypress` : lorsqu'une touche est pressée et relâchée
- (c) `onkeyup` : lorsqu'une touche est relâchée

### 3. Événements formulaire

- (a)
- (b) `onblur` : à la perte du focus
- (c) `onchange` : à la perte du focus si la valeur a changé
- (d) `onfocus` : lorsque l'élément prend le focus (ou devient actif)
- (e) `onreset` : lors de la remise à zéro du formulaire (via un bouton "reset" ou une fonction `reset()`)
- (f) `onselect` : quand du texte est sélectionné
- (g) `onsubmit` : quand le formulaire est validé (via un bouton de type "submit" ou une fonction `submit()`)

## B.2.2 Exemple de mise à jour d'un élément

Code Source B.3 : Génération et sortie du code *JSON* en *PHP*

```
1 <!doctype html>
2 <html lang="fr">
3   <head>
4     <meta charset="utf-8"/>
5     <link rel="stylesheet" href="./myStyle.css"/>
6     <style>
7       body{
8         font-family : "Comic Sans MS";
9         font-size : 120%;
10      }
11      h1{
12        margin : 0 auto;
13        text-align : center;
14      }
15    </style>
```



```

16 <title>Mise    Jour Par   v  nement</title>
17 </head>
18 <body>
19 <!-- d  but du corps HTML -->
20 <h1>Mise    Jour Par   v  nement <code>onchange</code></h1>
21 <p class="popupDetails">
22     <input id="myInputId" type="text" size="15"
23         onchange="fonctionMiseAJour( 'myInputContent', 'myInputId' )"/>
24     <br/>
25     <span id="myInputContent"></span>
26 </p>
27 <script>
28     function fonctionMiseAJour(elementId , inputId){
29         document.getElementById(elementId).innerHTML
30             = document.getElementById(inputId).value ;
31     }
32 </script>
33 </body>
34 <!-- fin du corps HTML -->
35 </html>
36 <!-- fin du code HTML -->

```

### B.2.3 Formulaire Dynamiques an *Javascript*

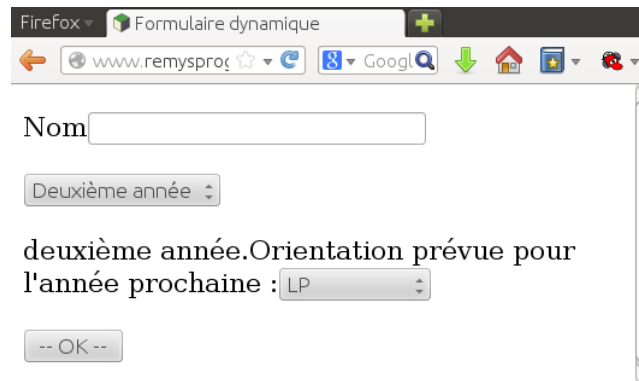
Nous voyons ici un exemple d'utilisation du *javascript* pour cr  er un formulaire dont les attributs d  pendent de la valeur d'un premier champ. Lorsqu'on s  lectionne "deuxi  me ann  e", un nouveau champ appara  t. Pour cela, on utilise l'  v  nement **onchange** sur l'**input** de l'ann  e, qui est g  r   par la fonction **anneeChange**. On teste alors la valeur de l'attribut, puis le cas   ch  ant on g  n  re un nouveau champ dans un **div** d'id **attributSupplementaire**.

Code Source B.4 : G  n  ration et sortie du code *JSON* en *PHP*

```

1 <!doctype html>
2 <html lang="fr">
3   <head>
4     <meta charset="UTF-8"/>
5     <title>Formulaire dynamique</title>
6   </head>
7   <body>
8     <form method="post" action="reception.php">
9       <p>
10        <label for="nom">Nom</label><input name="nom" id="nom"/>

```



```

11     </p>
12     <p>
13         <select name="annee" id="annee" pattern="(premiere) | (deuxieme)"
14             onchange='anneeChange();'>
15             <option value="choisissez" selected disabled>— choisissez —</option>
16             <option value="premiere">Première année</option>
17             <option value="deuxieme">Deuxième année</option>
18         </select>
19     </p>
20     <div id="attributSupplementaire">
21
22     </div>
23     <p>
24         <input type="submit" value="— OK —"/>
25     </p>
26 </form>
27 <script>
28     function anneeChange() {
29         var paragraphe = document.getElementById("attributSupplementaire");
30         paragraphe.innerHTML=document.getElementById("annee").value+" année.<br />";
31         if (document.getElementById("annee").value == "deuxieme"){
32             paragraphe.innerHTML+="

```

Code Source B.5 : Génération et sortie du code *JSON* en *PHP*

```

1 <!doctype html>
2 <html lang="fr">
3     <head>

```

```
4      <meta charset="UTF-8"/>
5      <title>Formulaire dynamique</title>
6  </head>
7  <body>
8  <?php
9      $nom= (isset($_POST["nom"])) ? $_POST["nom"] : "nom ind termin ";
10     $annee = (isset($_POST["annee"])) ? $_POST["annee"] : "ann e ind termin e";
11         echo "Nom : ".$nom."<br/>";
12         echo "Ann e : ".$annee."<br/>";
13     if ($annee=="deuxi me")
14         echo " Orientation : ".$_POST["orientation"];
15
16
17 ?>
18 </body>
19 </html>
```

# Annexe C

## Gestion des fenêtres

### C.1 Charger un nouveau document

Code Source C.1 : Génération et sortie du code *JSON* en *PHP*

```
1 // constructeur
2 function Telephone(tell){
3     // test de téléphone fran_ais à 10 chiffres
4     // 1) supprimer les espaces, 2) tester les chiffres
5     if (tell.replace(/\s/g, ' ').match(/^(|\+33|0)[0-9]{9}$/g))
6         this.tell=tell;
7     else
8         throw new Error("Numéro de téléphone invalide");
9 }
10
11 Telephone.prototype.affiche = function(){
12     document.write("Téléphone 1 : "+this.tell+"<br/>");
13 }
```

Code Source C.2 : Génération et sortie du code *JSON* en *PHP*

```
1 <!doctype HTML>
2 <html lang="fr">
3 <head>
4 <meta charset="UTF-8" />
5 <title>Charger un document</title>
6 <script src="./ex01_classes_telephone.js"></script>
7 </head>
8 <body>
9 <p>
10 <script>
11 try{
12     var numero = prompt("Merci d'entrer un numéro de téléphone en France mé
13         tropolitaine");
14     var tel = new Telephone(numero);
15     tel.affiche();
16 }catch (err){
17     location = "ex01_error.html";
18 }
19 </script>
20 <p>
```

```
20 </body>
21 </html>
```

Code Source C.3 : Génération et sortie du code *JSON* en *PHP*

```
1 <!doctype HTML>
2 <html lang="fr">
3 <head>
4 <meta charset="UTF-8" />
5 <title>Charger un document</title>
6 <script src="./ex10_class es_telephone.js"></script>
7 </head>
8 <body>
9 <p>
10 Bonjour, Il s'est produit une erreur. Merci d'entrer un numéro valide.
11 Si le problème persiste, merci de contacter le stagiaire qui a fait le site...
12 <button onclick="location = 'ex01_loadNewDoc.html';">Retour à la saisie</button>
13 <p>
14 </body>
15 </html>
```

## C.2 Naviguer dans l'historique

la propriété `history` a deux méthodes `back()` et `forward()` qui permettent respectivement de reculer ou d'avancer dans l'historique.

Code Source C.4 : Génération et sortie du code *JSON* en *PHP*

```
1 <!doctype HTML>
2 <html lang="fr">
3 <head>
4 <meta charset="UTF-8" />
5 <title>Charger un document</title>
6 <script src="./ex10_class es_telephone.js"></script>
7 </head>
8 <body>
9 <p>
10 Bonjour, bla, bla...<br/>
11 <a href = "ex02_historyBack.html">Cliquez ici</a> pour aller à la page suivante.
12 <p>
13 </body>
14 </html>
```

Code Source C.5 : Génération et sortie du code *JSON* en *PHP*

```
1 <!doctype HTML>
2 <html lang="fr">
3 <head>
4 <meta charset="UTF-8" />
5 <title>Charger un document</title>
6 <script src="./ex10_class es_telephone.js"></script>
7 </head>
8 <body>
9 <p>
```

```

10  Bla , bla ... <br/>
11  Vous avez raté quelque chose ?
12  <button onclick="history.back();">Retour à la page précédente</button>
13  <p>
14  </body>
15  </html>

```

## C.3 Ouvrir une nouvelle fenêtre (popup)

Code Source C.6 : Génération et sortie du code *JSON* en *PHP*

```

1  <!doctype HTML>
2  <html lang="fr">
3  <head>
4  <meta charset="UTF-8" />
5  <title>Ouvrir un fenêtre</title>
6  <script src="./ex10_classes_telephone.js"></script>
7  </head>
8  <body>
9  <p>
10  Bonjour , bla , bla ...
11  <button onclick="window.open('ex03_windowPopup.html', 'ma popup', 'width=400,
    height=400,resizeable=yes');">
12  Plus d'infos
13  </button>
14  <p>
15  </body>
16  </html>

```

Code Source C.7 : Génération et sortie du code *JSON* en *PHP*

```

1  <!doctype HTML>
2  <html lang="fr">
3  <head>
4  <meta charset="UTF-8" />
5  <title>Charger un document</title>
6  <script src="./ex10_classes_telephone.js"></script>
7  </head>
8  <body>
9  <p style="font-size : 100; text-align : center;">
10  Coucou !
11  <p>
12  <p>
13  <a href="javascript :window.close();">Fermer la fenêtre</a>
14  </p>
15  </body>
16  </html>

```

# Annexe D

## *Document Object Model (DOM)*

La programmation côté client permet de modifier certaines parties d'un document *HTML* dans recharger toute la page. Il y a plusieurs avantages : on évite de surcharger le serveur et le trafic réseau et on améliore la réactivité de l'application *web* pour le plus grand bonheur de l'utilisateur.

Pour faire cela, le langage *Javascript* côté client fournit une structure de données permettant d'accéder aux éléments du document *HTML* et de modifier les éléments du document *HTML*. Cette structure de données s'appelle le *Document Object Model*, en abrégé *DOM*. Il existe un *DOM* legacy qui s'est sédimenté informellement au travers des versions successives du *Javascript* en tenant compte des implémentations des différents navigateurs, qui collaboraient plus ou moins bien pour être mutuellement compatibles. Il existe aussi le *DOM* tel qu'il a été finalement spécifié par le *W3C*.

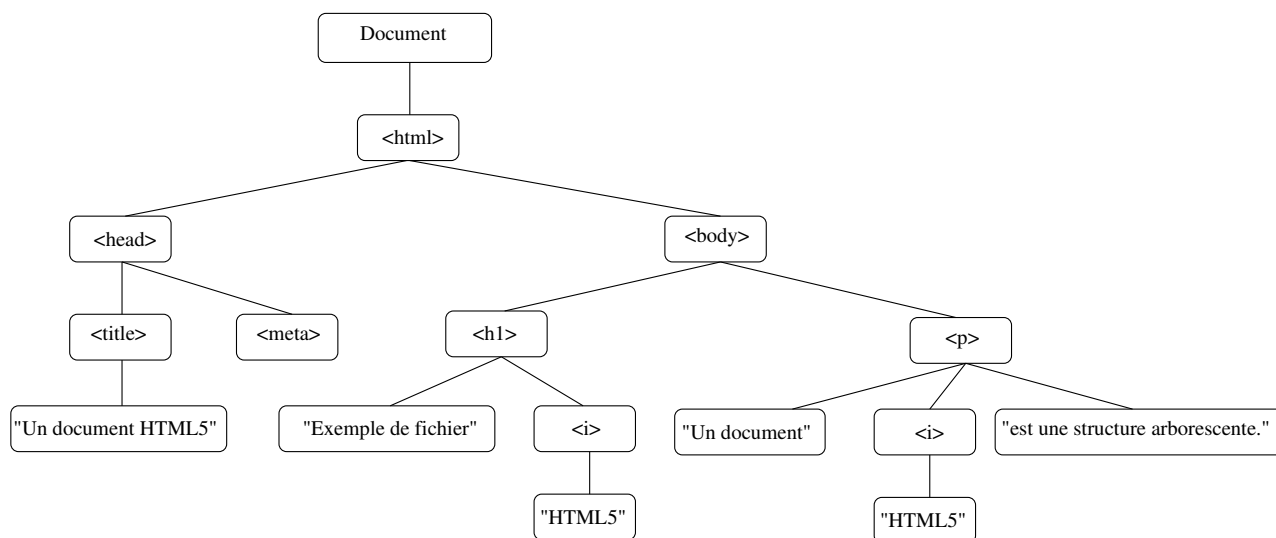
Les éléments du document *HTML* ayant, de par leur imbrication, une structure arborescente, le *DOM W3C* a une structure d'arbre. On peut accéder et manipuler via un ensemble de propriétés et de méthodes *Javascript*, notamment de l'interface **Document** et de l'interface **Element** et ses classes filles, qui permettent de manipuler les éléments (*HTML* entre autres) du document.

### D.1 Qu'est-ce que le *DOM* ?

Le *Document Object Model* (en abrégé *DOM*) correspond à l'arborescence des imbrications des balises *HTML* d'un document. Voici un fichier *HTML* simple et une représentation schématique du *DOM* correspondant.

Code Source D.1 : Génération et sortie du code *JSON* en *PHP*

```
1 <!doctype HTML>
2 <html lang="fr">
3 <head>
4 <meta charset="UTF-8" />
5 <title>Un document HTML5</title>
6 </head>
7 <body>
8   <h1>Exemple de fichier <i>HTML5</i></h1>
9   <p>
10     Un document <i>HTML5</i> est une structure arborescente.
11   <p>
12 </body>
```

13 `</html>`

Le *DOM* dont nous parlons ici est le *DOM* du *W3C*, qui est aujourd'hui supporté par tous les grands navigateurs.

Le langage *Javascript* côté client propose une hiérarchie de classes pour parcourir et manipuler le *DOM* d'un document. Il s'agit essentiellement d'une structure de donnée d'arbre, où chaque noeud (correspondant à une balise ou commentaire ou texte, etc. du document) possède une collection de noeuds fils, qui sont les éléments ou structures imbriquées.

La bibliothèque *jQuery* permet un accès plus haut niveau au *DOM* pour sonder et manipuler le code du document.

## D.2 Sélection et Manipulation de Base sur le *DOM*

### D.2.1 Sélection de tout ou partie des éléments

L'exemple suivant cherche tous les éléments du document et affiche leur nom de balise (**tagName** ou **nodeName**). On apprend aussi à ajouter du code *HTML* à l'intérieur d'un élément (au début ou à la fin).

L'exemple suivant montre comment sélectionner certains éléments du document, par nom de balise, classe *CSS*, etc. On apprend aussi à modifier des propriétés *CSS* des éléments.

Code Source D.2 : Génération et sortie du code *JSON* en *PHP*

```

1 <!doctype html>
2 <html lang="fr">
3 <head>
4   <meta charset="utf-8">
5   <title>Modifier le style de certains éléments</title>
6   <script src=". /jquery-1.10.2.js "></script>
7   <style>
8     p.myClass {
9       background-color : #ddd ;
10      padding : 10px ;
11    }
  
```





```

12     </style>
13 </head>
14
15 <body>
16 <h1>Modifier le style de certains éléments</h1>
17 <div>
18     <h2>Partie 1</h2>
19     <p>Ceci est le texte de la partie 1.</p>
20 </div>
21 <div>
22     <h2>Partie 2</h2>
23     <p class="myClass">Le texte de la partie 2 est différent.</p>
24 </div>
25 <script>
26 // Récupération d'éléments jQuery pour les balises <p> et <h2>
27 var elements = $( "p, h2" );
28 elements.css( "border", "2px solid" );
29 // Modification du style du titre <h1>
30 $( "h1" ).css( "text-align", "center" );
31 // Modification du style du (ou des) paragraphe(s) de la classe CSS myClass
32 $( "p.myClass" ).css( "border-radius", "20px" );
33 </script>
34 </body>

```

## D.2.2 Filtrage par le texte

L'exemple suivant montre comment sélectionner des éléments par mots du texte (sensible à la casse).

Code Source D.3 : Génération et sortie du code *JSON* en *PHP*

```

1 <!doctype html>
2 <html lang="fr">
3 <head>
4     <meta charset="utf-8">
5     <title>Filtrage du texte</title>
6     <script src="/jquery-1.10.2.js"></script>

```



```

7  <style>
8  p {
9      padding : 10px 0;
10 }
11 </style>
12 </head>
13 <body>
14 <h1>Filtrage du texte</h1>
15 <div>
16     <h2>Partie 1</h2>
17     <p>Ceci est le texte de la partie 1.</p>
18 </div>
19 <div>
20     <h2>Partie 2</h2>
21     <p>Le texte de la <em>partie 2</em> est différent.</p>
22 </div>
23 <script>
24     $( "p:contains('différent')" ).prepend('<strong>Ce paragraphe contient le mot
25         "différent"</strong>.<br/>').css( "background-color", "#ddd" );
26 </script>
27 </body>

```

### D.2.3 Application de Méthode aux éléments

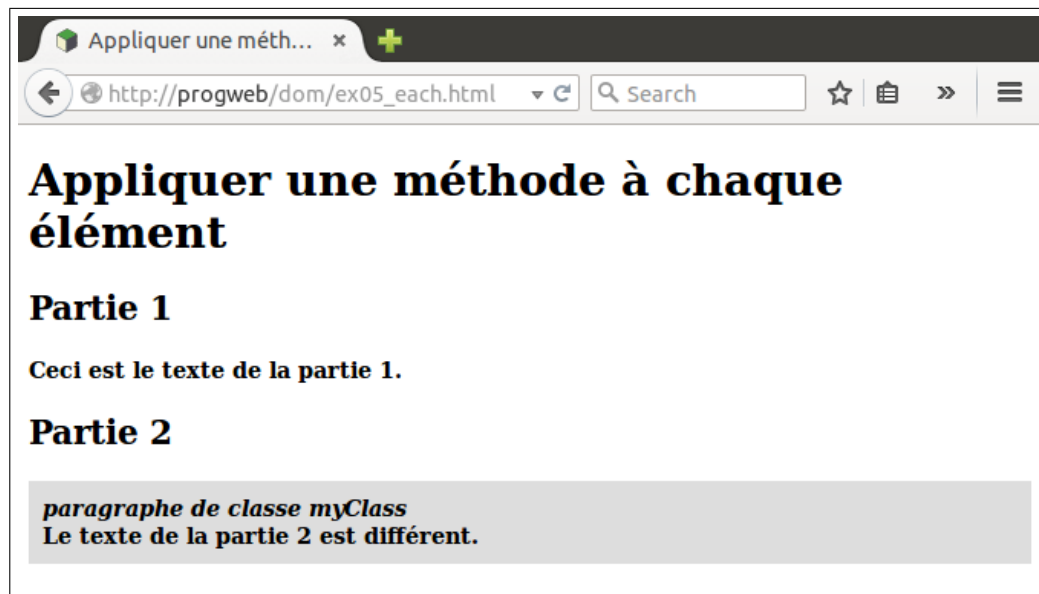
L'exemple suivant montre comment appliquer une fonction à chacun des éléments sélectionnés. Ici, on met le contenu des paragraphes en gras. On ajoute une information au début de chaque paragraphe de la classe `myClass`.

Code Source D.4 : Génération et sortie du code *JSON* en *PHP*

```

1  <!doctype html>
2  <html lang="fr">
3  <head>
4      <meta charset="utf-8">
5      <title>Appliquer une méthode à chaque élément</title>
6      <script src="/jquery-1.10.2.js"></script>

```



```

7  <style>
8  p.myClass {
9      background-color : #ddd;
10     padding : 10px;
11 }
12 </style>
13 </head>
14
15 <body>
16 <h1>Appliquer une méthode à chaque élément</h1>
17 <div>
18     <h2>Partie 1</h2>
19     <p>Ceci est le texte de la partie 1.</p>
20 </div>
21 <div>
22     <h2>Partie 2</h2>
23     <p class="myClass">Le texte de la partie 2 est différent.</p>
24 </div>
25 <script>
26     $( "p" ).each(function() {
27         $( this ).css( "font-weight", "bolder" );
28         if ( $( this ).hasClass( "myClass" )) {
29             $( this ).prepend( "<em>paragraphe de classe myClass</em><br />" );
30         }
31     });
32 </script>
33 </body>

```

## D.2.4 Événements et *Callbacks*

EL'exemple suivant montre comment, en réaction au click sur un bouton, transformer es paragraphes en div.

Code Source D.5 : Génération et sortie du code *JSON* en *PHP*



```

1 <!doctype html>
2 <html lang="fr">
3 <head>
4   <meta charset="utf-8">
5   <title>Événement de click</title>
6   <script src="/jquery-1.10.2.js"></script>
7   <style>
8     p {
9       background-color : #ddd;
10      padding : 10px;
11    }
12    div.myClass {
13      font-weight : bolder;
14      padding : 10px;
15      border-style : dashed;
16    }
17    em {
18      font-variant : small-caps;
19      font-size : 120%;
20    }
21    button {
22      margin : 10px 0;
23    }
24  </style>
25 </head>
26
27 <body>
28 <h1>Événement de click</h1>
29 <div>
30   <h2>Partie 1</h2>
31   <p>Ceci est le texte de la partie 1.</p>
32 </div>
33 <div>
34   <h2>Partie 2</h2>
35   <p>Le texte de la <em>partie 2</em> est différent.</p>

```

```

36 </div>
37 <button>Modifier les paragraphes</button>
38 <script>
39   // Événement de click
40   $( "button" ).click( function() {
41       // Application d'une méthode à chaque paragraphe
42       $( "p" ).each( function() {
43           // Remplacer le <p> par un <div> en laissant le HTML inchangé
44           $( this ).replaceWith( '<div class="myClass">' + $( this ).html()
45                                   + "</div>" );
46       } );
47   } );
48 </script>
</body>

```

## D.2.5 Filtrage d'un Tableau

L'exemple suivant montre comment, en utilisant les utilitaires de *jQuery* permettant de traiter des *Array Javascript* génériques :

1. Filtrer le contenu d'un tableau avec une méthode de choix booléenne pour les éléments (ici, valeur multiple de 3) ;
2. Générer le *HTML* en appliquant une méthode à chaque élément du tableau.



Code Source D.6 : Génération et sortie du code *JSON* en *PHP*

```

1 <!doctype HTML>
2 <html lang="fr">
3 <head>
4 <meta charset="UTF-8" />
5 <title>Filtrage Grep sur Tableau</title>
6 </head>

```

```
7 <body>
8   <h1>Filtrage Grep sur Tableau</h1>
9   <script src= ". /jquery-1.10.2.js ">/script>
10
11 <p id= "output ">
12
13 <script>
14   // Création d'un tableau avec les entiers de 0 à 19
15   var tab = new Array();
16   for (var i=0 ; i<20 ; i++){
17     tab.push(i);
18   }
19
20   // Sélection des éléments du tableau par la fonction "multiple de 3"
21   var tabMultipleDe3 = $.grep(tab, function(key, value){
22     if (key%3 == 0)
23       return true;
24     else
25       return false;
26   });
27
28   // Affichage du tableau des multiples de 3
29   var outHTML = "";
30   // Application d'une fonction (génération d'HTML)
31   // à chaque élément du tableau
32   $.each(tabMultipleDe3, function(key, value){
33     outHTML += "tab["+key+"] = "+value+"<br/>";
34   });
35   $( "#output" ).append( outHTML );
36 </script>
37 </body>
38 </html>
```